

Effective Theories in Programming Practice

Jayadev Misra



ASSOCIATION FOR COMPUTING MACHINERY

Effective Theories in Programming Practice

ACM Books

Editors in Chief

Sanjiva Prasad, *Indian Institute of Technology (IIT) Delhi, India*

Marta Kwiatkowska, *University of Oxford, UK*

Charu Aggarwal, *IBM Corporation, USA*

ACM Books is a new series of high-quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

Spatial Gems, Volume 1

Editors: John Krumm, *Microsoft Research, Microsoft Corporation, Redmond, WA, USA*

Andreas Züfle, *Geography and Geoinformation Science Department, George Mason University, Fairfax, VA, USA*

Cyrus Shahabi, *Computer Science Department, University of Southern California, Los Angeles, CA, USA*

2022

Edsger Wybe Dijkstra: His Life, Work, and Legacy

Editors: Krzysztof R. Apt, *CWI, Amsterdam and University of Warsaw*

Tony Hoare, *University of Cambridge and Microsoft Research Ltd*

2022

Weaving Fire into Form: Aspirations for Tangible and Embodied Interaction

Brygg Ullmer, *Clemson University*

Orit Shaer, *Wellesley College*

Ali Mazalek, *Toronto Metropolitan University*

Caroline Hummels, *Eindhoven University of Technology*

2022

Linking the World's Information: A Collection of Essays on the Work of Sir Tim Berners-Lee

Oshani Seneviratne, *Rensselaer Polytechnic Institute*

James A. Hendler, *Rensselaer Polytechnic Institute*

2022

Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman

Editor: Rebecca Slayton, *Cornell University*

2022

Applied Affective Computing

Leimin Tian, *Monash University*

Sharon Oviatt, *Monash University*
Michal Muszynski, *Carnegie Mellon University and University of Geneva*
Brent C. Chamberlain, *Utah State University*
Jennifer Healey, *Adobe Research, San Jose*
Akane Sano, *Rice University*
2022

Circuits, Packets, and Protocols: Entrepreneurs and Computer Communications, 1968–1988

James L. Pelkey
Andrew L. Russell, *SUNY Polytechnic Institute, New York*
Loring G. Robbins
2022

Theories of Programming: The Life and Works of Tony Hoare

Editors: Cliff B. Jones, *Newcastle University, UK*
Jayadev Misra, *The University of Texas at Austin, US*
2021

Software: A Technical History

Kim W. Tracy, *Rose-Hulman Institute of Technology, IN, USA*
2021

**The Handbook on Socially Interactive Agents: 20 years of Research on Embodied Conversational Agents, Intelligent Virtual Agents, and Social Robotics
Volume 1: Methods, Behavior, Cognition**

Editors: Birgit Lugin, *Julius-Maximilians-Universität of Würzburg*
Catherine Pelachaud, *CNRS-ISIR, Sorbonne Université*
David Traum, *University of Southern California*
2021

Probabilistic and Causal Inference: The Works of Judea Pearl

Editors: Hector Geffner, *ICREA and Universitat Pompeu Fabra*
Rina Dechter, *University of California, Irvine*
Joseph Y. Halpern, *Cornell University*
2022

Event Mining for Explanatory Modeling

Laleh Jalali, *University of California, Irvine (UCI), Hitachi America Ltd.*
Ramesh Jain, *University of California, Irvine (UCI)*
2021

Intelligent Computing for Interactive System Design: Statistics, Digital Signal Processing, and Machine Learning in Practice

Editors: Parisa Eslambolchilar, *Cardiff University, Wales, UK*
Andreas Komninos, *University of Patras, Greece*
Mark Dunlop, *Strathclyde University, Scotland, UK*
2021

Semantic Web for the Working Ontologist: Effective Modeling for Linked Data, RDFS, and OWL, Third Edition

Dean Allemang, *Working Ontologist LLC*

Jim Hendler, *Rensselaer Polytechnic Institute*

Fabien Gandon, *INRIA*

2020

Code Nation: Personal Computing and the Learn to Program Movement in America

Michael J. Halvorson, *Pacific Lutheran University*

2020

Computing and the National Science Foundation, 1950–2016: Building a Foundation for Modern Computing

Peter A. Freeman, *Georgia Institute of Technology*

W. Richards Adrion, *University of Massachusetts Amherst*

William Aspray, *University of Colorado Boulder*

2019

Providing Sound Foundations for Cryptography: On the work of Shafi Goldwasser and Silvio Micali

Oded Goldreich, *Weizmann Institute of Science*

2019

Concurrency: The Works of Leslie Lamport

Dahlia Malkhi, *VMware Research* and *Calibra*

2019

The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!

Ivar Jacobson, *Ivar Jacobson International*

Harold “Bud” Lawson, *Lawson Konsult AB (deceased)*

Pan-Wei Ng, *DBS Singapore*

Paul E. McMahon, *PEM Systems*

Michael Goedicke, *Universität Duisburg–Essen*

2019

Data Cleaning

Ihab F. Ilyas, *University of Waterloo*

Xu Chu, *Georgia Institute of Technology*

2019

Conversational UX Design: A Practitioner’s Guide to the Natural Conversation Framework

Robert J. Moore, *IBM Research–Almaden*

Raphael Arar, *IBM Research–Almaden*

2019

[Heterogeneous Computing: Hardware and Software Perspectives](#)

Mohamed Zahran, *New York University*

2019

[Hardness of Approximation Between P and NP](#)

Aviad Rubinstein, *Stanford University*

2019

[The Handbook of Multimodal-Multisensor Interfaces, Volume 3: Language Processing, Software, Commercialization, and Emerging Directions](#)

Editors: Sharon Oviatt, *Monash University*

Björn Schuller, *Imperial College London and University of Augsburg*

Philip R. Cohen, *Monash University*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*

2019

[Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker](#)

Editor: Michael L. Brodie, *Massachusetts Institute of Technology*

2018

[The Handbook of Multimodal-Multisensor Interfaces, Volume 2: Signal Processing, Architectures, and Detection of Emotion and Cognition](#)

Editors: Sharon Oviatt, *Monash University*

Björn Schuller, *University of Augsburg and Imperial College London*

Philip R. Cohen, *Monash University*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*

2018

[Declarative Logic Programming: Theory, Systems, and Applications](#)

Editors: Michael Kifer, *Stony Brook University*

Yanhong Annie Liu, *Stony Brook University*

2018

[The Sparse Fourier Transform: Theory and Practice](#)

Haitham Hassanieh, *University of Illinois at Urbana-Champaign*

2018

[The Continuing Arms Race: Code-Reuse Attacks and Defenses](#)

Editors: Per Larsen, *Immunant, Inc.*

Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

2018

Frontiers of Multimedia Research

Editor: Shih-Fu Chang, *Columbia University*

2018

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable

Julian Shun, *University of California, Berkeley*

2017

Computational Prediction of Protein Complexes from Protein Interaction Networks

Sriganesh Srihari, *The University of Queensland Institute for Molecular Bioscience*

Chern Han Yong, *Duke-National University of Singapore Medical School*

Limsoon Wong, *National University of Singapore*

2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations

Editors: Sharon Oviatt, *Incaa Designs*

Björn Schuller, *University of Passau and Imperial College London*

Philip R. Cohen, *Voicebox Technologies*

Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*

Gerasimos Potamianos, *University of Thessaly*

Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*

2017

Communities of Computing: Computer Science and Society in the ACM

Thomas J. Misa, Editor, *University of Minnesota*

2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining

ChengXiang Zhai, *University of Illinois at Urbana-Champaign*

Sean Massung, *University of Illinois at Urbana-Champaign*

2016

An Architecture for Fast and General Data Processing on Large Clusters

Matei Zaharia, *Stanford University*

2016

Reactive Internet Programming: State Chart XML in Action

Franck Barbier, *University of Pau, France*

2016

Verified Functional Programming in Agda

Aaron Stump, *The University of Iowa*

2016

[The VR Book: Human-Centered Design for Virtual Reality](#)

Jason Jerald, *NextGen Interactions*

2016

[Ada's Legacy: Cultures of Computing from the Victorian to the Digital Age](#)

Robin Hammerman, *Stevens Institute of Technology*

Andrew L. Russell, *Stevens Institute of Technology*

2016

[Edmund Berkeley and the Social Responsibility of Computer Professionals](#)

Bernadette Longo, *New Jersey Institute of Technology*

2015

[Candidate Multilinear Maps](#)

Sanjam Garg, *University of California, Berkeley*

2015

[Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing](#)

John Cullinane, *Northeastern University; Mossavar-Rahmani Center for Business and Government, John F. Kennedy School of Government, Harvard University*

2015

[A Framework for Scientific Discovery through Video Games](#)

Seth Cooper, *University of Washington*

2014

[Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers](#)

Bryan Jeffrey Parno, *Microsoft Research*

2014

[Embracing Interference in Wireless Systems](#)

Shyamnath Gollakota, *University of Washington*

2014

Effective Theories in Programming Practice

Jayadev Misra

The University of Texas at Austin, US

ACM Books #47



Copyright © 2022 by Association for Computing Machinery

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews—without the prior permission of the publisher.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which the Association of Computing Machinery is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Effective Theories in Programming Practice

Jayadev Misra

books.acm.org

<http://books.acm.org>

ISBN: 978-1-4503-9973-9 hardcover

ISBN: 978-1-4503-9971-5 paperback

ISBN: 978-1-4503-9972-2 EPUB

ISBN: 978-1-4503-9974-6 eBook

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3568325	Book	10.1145/3568325.3568331	Chapter 5
10.1145/3568325.3568326	Preface	10.1145/3568325.3568332	Chapter 6
10.1145/3568325.3568327	Chapter 1	10.1145/3568325.3568333	Chapter 7
10.1145/3568325.3568328	Chapter 2	10.1145/3568325.3568334	Chapter 8
10.1145/3568325.3568329	Chapter 3	10.1145/3568325.3568335	Bibliography
10.1145/3568325.3568330	Chapter 4	10.1145/3568325.3568336	Bios/Index

A publication in the ACM Books series, #47

Editors in Chief: Sanjiva Prasad, *Indian Institute of Technology (IIT) Delhi, India*

Marta Kwiatkowska, *University of Oxford, UK*

Charu Aggarwal, *IBM Corporation, USA*

Area Editors: M. Tamer Özsu, *University of Waterloo*

Sanjiva Prasad, *Indian Institute of Technology (IIT) Delhi, India*

This book was typeset in Arnheim Pro 10/14 and Flama using pdf_T_EX.

First Edition

10 9 8 7 6 5 4 3 2 1

In memory of Edsger W. Dijkstra
colleague, friend, mentor

Contents

Preface xvii

Acknowledgment xx

Chapter 1 Introduction 1

- 1.1 Motivation for This Book 1
- 1.2 Lessons from Programming Theory 2
- 1.3 Formalism: The Good, the Bad and the Ugly 4

Chapter 2 Set Theory, Logic and Proofs 7

- 2.1 Set 7
- 2.2 Function 17
- 2.3 Relation 25
- 2.4 Order Relations: Total and Partial 32
- 2.5 Propositional Logic 37
- 2.6 Predicate Calculus 50
- 2.7 Formal Proofs 54
- 2.8 Examples of Proof Construction 59
- 2.9 Exercises 74

Chapter 3 Induction and Recursion 83

- 3.1 Introduction 83
- 3.2 Examples of Proof by Induction 87
- 3.3 Methodologies for Applying Induction 93
- 3.4 Advanced Examples 101
- 3.5 Noetherian or Well-founded Induction 115
- 3.6 Structural Induction 122
- 3.7 Exercises 125

Chapter 4 Reasoning About Programs 139

- 4.1 Overview 139

4.2	Fundamental Ideas	141
4.3	Formal Treatment of Partial Correctness	145
4.4	A Modest Programming Language	148
4.5	Proof Rules	152
4.6	More on Invariant	164
4.7	Formal Treatment of Termination	171
4.8	Reasoning about Performance of Algorithms	175
4.9	Order of Functions	178
4.10	Recurrence Relations	182
4.11	Proving Programs in Practice	190
4.12	Exercises	193
4.A	Appendix: Proof of Theorem 4.1	200
4.B	Appendix: Termination in Chameleons Problem	201

Chapter 5 Program Development 203

5.1	Binary Search	203
5.2	Saddleback Search	205
5.3	Dijkstra's Proof of the am-gm Inequality	207
5.4	Quicksort	208
5.5	Heapsort	215
5.6	Knuth-Morris-Pratt String-matching Algorithm	222
5.7	A Sieve Algorithm for Prime Numbers	233
5.8	Stable Matching	245
5.9	Heavy-hitters: A Streaming Algorithm	250
5.10	Exercises	256

Chapter 6 Graph Algorithms 265

6.1	Introduction	265
6.2	Background	266
6.3	Specific Graph Structures	274
6.4	Combinatorial Applications of Graph Theory	281
6.5	Reachability in Graphs	289
6.6	Graph Traversal Algorithms	292
6.7	Connected Components of Graphs	304
6.8	Transitive Closure	317
6.9	Single Source Shortest Path	325
6.10	Minimum Spanning Tree	339
6.11	Maximum Flow	349
6.12	Goldberg-Tarjan Algorithm for Maximum Flow	356
6.13	Exercises	364

- 6.A Appendix: Proof of the Reachability Program 371
- 6.B Appendix: Depth-first Traversal 373

Chapter 7 Recursive and Dynamic Programming 377

- 7.1 What is Recursive Programming 377
- 7.2 Programming Notation 379
- 7.3 Reasoning About Recursive Programs 406
- 7.4 Recursive Programming Methodology 413
- 7.5 Recursive Data Types 424
- 7.6 Dynamic Programming 432
- 7.7 Exercises 445
- 7.A Appendices 453

Chapter 8 Parallel Recursion 459

- 8.1 Parallelism and Recursion 459
- 8.2 Powerlist 460
- 8.3 Pointwise Function Application 464
- 8.4 Proofs 466
- 8.5 Advanced Examples Using Powerlists 474
- 8.6 Multi-dimensional Powerlist 492
- 8.7 Other Work on Powerlist 500
- 8.8 Exercises 501
- 8.A Appendices 504

Bibliography 511

Author's Biography 519

Index 521

Preface

“He who loves practice without theory is like the sailor who boards a ship without a rudder and compass and never knows where he may be cast.”

Leonardo da Vinci

“The distance between theory and practice is always smaller in theory than in practice.”

Marshall T. Rose

What this book is about This book explains a number of fundamental algorithms in a concise, yet precise, manner. It includes the background material needed to understand the explanations and to develop such explanations for other algorithms. What students prize most in an algorithm explanation are clarity and simplicity. The thesis of this book is that clarity and simplicity are achieved not by eschewing formalism but by suitably using it.

I have taught the topics in this book several times to undergraduates in their sophomore and junior years at the University of Texas at Austin. Though many students struggled, never having used any formal reasoning, most got something out of the course and many truly excelled. Anecdotal evidence from the instructors of subsequent courses suggests that students who had taken the course generally did better than the ones who had not.

The professed goal of every educator is to prepare the students for the future. Since it is harder to predict the future than the past, it is safe to assume that certain topics—set theory, logic, discrete mathematics and fundamental algorithms along with their correctness and complexity analysis—will always remain useful for computing professionals. The book covers these materials but includes many other topics that will stimulate the students.

What this book is not This book is not a compendium on any particular area. This is not a book on discrete mathematics that is typically taught during the first two years of an undergraduate curriculum, though many of the topics covered in

those courses, such as set theory and elementary logic, are covered in this book as background material.

This book is not about data structures and their associated algorithms. It does not cover many fundamental algorithms, such as sorting, searching, parsing and database algorithms, which are often required material for a computer science degree.

In the courses I have taught, I covered several application areas where I could illustrate the use of formal methods developed here. Among them: data compression, error-detecting and error-correcting codes, cryptography, and finite state machines. I do not include them here because of space limitations (the size of the book) and time limitations (my own). This book does not touch on many important theories, such as formal grammars, database and security. I do not cover asynchronous concurrency, my own area of research, or real-time programs.

Background required for the book The book assumes only a background in high school mathematics and some elementary program writing skills. It is largely self-contained. No production programming language is used in the book. A modest abstract programming language is introduced that can be understood by any one with knowledge of imperative programming. Recursive programming is explained in Chapter 7 where an elementary version of the programming language Haskell is introduced.

The structure of the book The introductory chapter, Chapter 1, explains the motivation and philosophy of the book because the included material may seem idiosyncratic. Chapter 2 covers basic set theory, functions and relations, propositional logic, and elementary predicate calculus. This chapter goes considerably beyond the topics covered in high school. The chapter on induction, Chapter 3, again goes much deeper than the elementary induction principle. Chapter 4, on program correctness and complexity analysis, explains the theory of inductive assertion method and techniques for solving recurrence relations. Chapter 5 applies program analysis techniques to several problems of moderate complexity. Chapter 6 contains basic material on graphs and a few fundamental algorithms. Recursive programs are explained in great detail in Chapters 7 and 8.

How to use the book

Reading the book casually To get a sense of the book, scan over the table of contents first. Those who believe they know the material should still look at Chapter 2 for the notations introduced for quantified expressions and the style adopted for writing formal proofs. Chapter 4 on program verification introduces the style adopted

for writing formal program proofs. The reader can then choose the sections to read in any order.

Studying the book The book is designed for self-study. It is complete, all the needed background is given, and the explanations are never skimpy. I recommend reading the book slowly, perhaps in a printed version; the digital version has advantages in that a reader can navigate using hyperlinks and distinguish between proofs and program code that use different colors. On first reading skip the sections that claim to contain advanced material. I expect that a motivated student who goes through the text carefully, studies the examples and works out the exercises can learn it without supervision.

Teaching from the book I have taught some of the material in a one-semester course to sophomores and juniors, though there is too much material in the book for such a course. For a one-semester course I recommend Chapters 2, 3 and 4, avoiding the sections that contain advanced material. The teacher may then choose a subset of examples from Chapter 5 or delve into Chapter 6 on graph theory. For a course on programming theory and methodology, the chapter on graph theory may be replaced by the chapter on sequential recursion, Chapter 7. The chapter on parallel recursion, Chapter 8, is more appropriate for a graduate class or seminar.

Terminology and conventions used in the book I use both the terms “I” and “we” in the book with different meanings. The term “I” refers to me as the author of the book, as in the sentence: “I will now show two proofs of this theorem.” The term “we” is not a royal version of “I” but refers to me and the readers whom I invite to join me in a collective effort, as in: “we have now seen two proofs of the theorem.”

I use “iff” as an abbreviation for “if and only if” throughout the book. The common mathematical abbreviations “at least” and “at most” are used, respectively, in place of greater than or equal to, $\geq$, and less than or equal to, $\leq$; so “ x is at least y ” means $x \geq y$.

In retrospect There is no acid test for evaluating the quality of explanation of an algorithm. I often cringe when I look at my own explanations that were written many years back. I hope I will have happier emotions when I read this book a decade or so from now.

I have not tested any of the imperative programs, but I have tested all the functional programs written in Haskell in Chapter 7. In spite of my best efforts, there may still be errors in the programs and in the text, some of the explanations may be longer than necessary, and it may be possible to shorten and improve some explanations by introducing better notations and formalisms. I would like to hear the readers’ suggestions.

Acknowledgment

I owe much to Edsger W. Dijkstra, in particular by engaging in acrimonious debates about even the most trifling scientific matters. Tony Hoare has been a friend and mentor who has unstintingly given me superb scientific advice throughout my career.

I am grateful to David Gries, Rick Hehner, S. Rao Kosaraju and Douglas McIlroy who not only read the manuscript for errors but suggested numerous improvements in style and substance. Their suggestions have led to several significant improvements. I am thankful to Vladimir Lifschitz who answered my questions about algebra and logic promptly and at a level that I can understand, and Scott Aaronson, who tried to enlighten me about the current state of quantum computing, sadly, without much success. Mani Chandy, a lifelong friend and a research inspiration to me, helped me with the chapter on graph theory.

The COVID pandemic has not been kind, particularly to my wife who has to put up with a house-bound husband all day. Thank you Mamata for your patience and help.

The original inspiration to design a computer science course that illustrates the applications of effective theories in practice came from Elaine Rich and J Moore. Mohamed Gouda and Greg Plaxton have used my original class notes to teach several sections of an undergraduate course; I am grateful to them for their suggestions. Several graduate teaching assistants, especially Xiaozhou (Steve) Li and Thierry Joffrain, have helped me revise my original notes.

My sincere thanks to the ACM editors and production staff, in particular Karen Grace, Sean Pidgeon and Bernadette Shade, who have helped immensely in producing the final manuscript.

Austin, Texas
December 22, 2022

Jayadev Misra

Introduction

1.1

Motivation for This Book

I am looking at the description of a classic algorithm in a classic book, an algorithm for computing the maximum flow in a network, in a book by [Ford and Fulkerson \[1962\]](#). The description of the algorithm followed the best practices of the day. First, an underlying theorem, the max-flow min-cut theorem, is stated and proved. Next, the algorithm is described in detail using labels on the edges of a graph. By current standards we can say that the algorithm description was primitive. Today we may describe the algorithm in a more structured manner, perhaps writing the main program in an abstract fashion (it is not important to understand the algorithm for what I have to say next):

```
start with any initial valid flow;
while there is a flow augmenting path do
    increase the flow along that path
enddo
```

Next, we may write a separate description of how to initialize, detect a flow augmenting path, and increase the flow along it. We prefer such a description because it explains the algorithm hierarchically, first showing the overall structure and then the details of the parts. We can choose to study one part carefully or ignore it because we may be familiar with it. We can continue the hierarchical description for as many levels as needed to fully explain the program.

The point of this example is merely to observe that we have learned a great deal about describing and explaining algorithms in the last 50 years. We know how the use of proper abstractions and structuring of an algorithm can dramatically simplify its explanation. The abstract program for maximum flow, even at this stage, imposes proof obligations: (1) if there is a flow augmenting path, the flow can be increased along it, and (2) if there is no flow augmenting path, the flow is maximum. We now have the mathematical tools for proving correctness of programs.

Given these advances, one would expect that descriptions of algorithms in modern textbooks would follow these practices. I am disappointed that explanations of algorithms are not significantly better today than they were 50 years ago. The advances made in programming theory have had minimal impact on how algorithms are explained to students. Most textbooks do not emphasize structured descriptions or explanations of algorithms. Their main concern is the efficiency of the algorithm and its implementation. Correctness issues are merely argued, relying on the reader's intuition and commonsense reasoning. Such arguments often refer to the step-by-step operation of the algorithm, forcing the reader to simulate the execution to gain an understanding. Only rarely does a description even mention an *invariant*, the main tool in proving correctness of programs and understanding their dynamic behaviors.

This book is an experiment on my part to explain, using the best practices of today, a number of sequential algorithms that I have admired over the years. In part, my effort is to refute the argument that careful and formal treatments of algorithms are too long and dense for students to appreciate.

1.2 Lessons from Programming Theory

Edsger Dijkstra once observed that humans and computers are both symbol processing systems, and there is a dividing line in software development where humans are more effective on one side of the line and the machines on the other. The goal of programming research is to move the line so that machines become effective on more aspects of programming.

In the last five decades research in programming theory has indeed moved the line significantly. Specifically, we have moved away from coding very specific machine details to using high-level programming languages, using advanced control and data structures including recursion, and applying static type checking, as exemplified in ML and Haskell, which permit only type-safe programs to be executed. And very significantly, we now have proof theories to establish correctness of an algorithm with respect to a given specification.

Advances in programming theory have made it possible to describe algorithms in a clear and precise fashion. *Hierarchical refinement* is perhaps the most important tool in program explanation. Start with an abstract description of the algorithm. Such a description will typically include *uninterpreted* components in control and data structures, as I have done for the maximum flow algorithm in the opening paragraph of this chapter. The description shows the overall structure of the algorithm but ignores the details of implementation of the components. The uninterpreted components, in this case “start with any initial valid flow”, “there is a flow augmenting path” and “increase the flow along that path”, are meaningful

only to humans if accompanied by further explanation. Given the mathematical specifications of the original problem and those of its components, the program can be proved correct.

At a later stage, we refine the components by adding details. The refinements may introduce further uninterpreted components. A formidable tool in concise description is recursion, both for control and data. The refinements continue to a point where the reader can supply the details for a complete implementation.

There is nothing new in my approach to algorithm description. Edsger Dijkstra was a pioneer in the field, see his book [Dijkstra \[1976\]](#) and numerous handwritten notes, called EWDs, in which he propounds this approach. One of the earliest books to apply Dijkstra's approach is [Gries \[1981\]](#). [Hehner \[1993, 2022\]](#) and [Morgan \[1990\]](#) not only apply the theory but make significant improvements in theory, notations and methodology.

Dijkstra showed that many parts of algorithms can be derived systematically from their specifications using a small set of heuristics. This book does not pretend to teach algorithm derivation. I have deliberately restricted myself to explaining existing algorithms, not deriving them. Thus, my emphasis is different from Dijkstra's.

I adopt semiformal explanations of algorithms, but I strive for precision in the description at all points. Precision does not mean everything is written down in a formal notation, only that descriptions are clear enough to convince most readers without undue mental effort on their part. Clarity may be achieved by the suitable use of formalism, but formalism is introduced only when it adds to clarity.

1.2.1 Terminology: Algorithms versus Programs

Often, a distinction is made between algorithms and programs, that an algorithm is an abstract description of a solution to a programming problem and a program is the actual artifact that can be executed on a computer. Given that algorithms are refined over multiple levels, and that the program is just the lowest level, I abandon this distinction and use both terms interchangeably.

1.2.2 Domain Knowledge

We expect a history teacher to know how to teach, but more importantly, know history. An elegant teaching style is no excuse for ignorance of the subject matter. Programming needs domain knowledge at every step of refinement. For many simple programs such knowledge may be just high school mathematics, so it is not explicitly acknowledged. In more advanced problems, say in designing software for communicating with a Mars rover, a detailed knowledge of coding theory and data compression is essential. In hierarchical program development, each refinement

requires specific domain knowledge. I introduce and develop domain knowledge, as needed, throughout this book, often designing a small theory for the solution of a specific problem.

1.3 Formalism: The Good, the Bad and the Ugly

Formalism typically evokes two kinds of reactions among students, extreme revulsion or sheepish acceptance. There is much discussion among curriculum developers about how much formalism computer science students need, if any. I have heard many arguments as to why students should learn new computer science material and not old mathematical material. I list the main themes of these arguments.

- (1) A sociological argument that many students, particularly the average ones, will not care for this type of explanation. And many are simply too afraid of formalism. So, to retain readership (and book sales), and to teach more effectively, eschew formalism.
- (2) Many readers of a book on algorithms are practicing programmers. They simply want to use the algorithms, not learn why they work. They assume that what is written in a book is correct, so they just want to copy the code and run it.
- (3) The correctness of a particular algorithm is easy to see; why belabor it.
- (4) Computer science is not mathematics, it is engineering. These ideas are too mathematical. Instead, we should emphasize the execution speed.

As a matter of fact, I am sympathetic to some of these arguments opposing formalism because, quite often, the formalism employed is baroque; it neither clarifies nor reduces the mental effort for the reader. Bad formalisms are simply wrong and the ugly ones cause revulsion in everyone. They are formal looking in having a number of mathematical symbols, though the formalism is never actually applied. Good formalisms take time to develop, teach and learn. They simplify our thinking and save us time.

1.3.1 Coxeter's Rabbit¹

I choose an example from [Dijkstra \[2002\]](#), the last in the EWD series, which contains a beautiful proof of an identity used in plane geometry. Dijkstra wrote this

1. For Dijkstra, a “rabbit”, as in a magician pulling a rabbit out of a hat, represented a conjuring trick in scientific matters where a seemingly clever result is shown without adequate explanation. He detested such rabbits.

note while he was gravely ill, a few months before his death. The point of including this proof here is to show how an appropriate formalism can vastly simplify a mathematical argument.

H.S.M. Coxeter [1961] includes, without proof, the following identity in connection with computing the area of a triangle of sides a , b and c : Given $s = (a + b + c)/2$,

$$s \cdot (s - b) \cdot (s - c) + s \cdot (s - c) \cdot (s - a) + s \cdot (s - a) \cdot (s - b) - (s - a) \cdot (s - b) \cdot (s - c) = a \cdot b \cdot c$$

The obvious way to prove is to replace s by $(a + b + c)/2$ in the left-hand side (lhs) of the identity and then simplify it to $a \cdot b \cdot c$ in the right-hand side (rhs). Dijkstra's proof is based on polynomials. Both lhs and rhs can be regarded as polynomials in a , b , c and s . Each of the factors in the lhs, s , $(s - a)$, $(s - b)$, $(s - c)$, is a linear combination of variables a , b and c with non-zero coefficients, and each term in the lhs, such as $s \cdot (s - b) \cdot (s - c)$ or $(s - a) \cdot (s - b) \cdot (s - c)$, is a product of exactly *three* factors. Therefore, each term is a polynomial of degree 3, so the lhs itself is a polynomial of degree 3. The rhs is also a polynomial of degree 3. Dijkstra shows that the rhs polynomial divides the lhs polynomial, and the quotient of division is 1.

Dijkstra exploits the symmetry of both sides in a , b and c to further simplify the proof: show only that the lhs is divisible by c ; then, by symmetry, it is divisible by a and b as well, so by $a \cdot b \cdot c$. The proof that the lhs is a multiple of c is quite simple. Consider the first two terms: $s \cdot (s - b) \cdot (s - c) + s \cdot (s - c) \cdot (s - a) = s \cdot (s - c) \cdot (s - b + s - a)$. Using $s = (a + b + c)/2$, $(s - b + s - a)$ is just c ; so $s \cdot (s - b) \cdot (s - c) + s \cdot (s - c) \cdot (s - a)$ is a multiple of c . Similarly, the last two terms $s \cdot (s - a) \cdot (s - b) - (s - a) \cdot (s - b) \cdot (s - c)$ is also a multiple of c .

Next, he determines the coefficient of multiplicity by computing the value of the lhs for one set of values of a , b and c , specifically, $a, b, c = 2, 2, 2$, which yields a coefficient of 1. So, the lhs equals $a \cdot b \cdot c$.

The use of polynomials and symmetry is what simplifies the proof. Someone who does not know about polynomials would not understand the proof. One goal of this book is to develop appropriate domain knowledge for a problem so that our explanation would be simpler than one just using common sense.

1.3.2 Why Use Formalism

People in favor of formalism often argue that it lends precision to the arguments, which is undoubtedly true. I believe, moreover, that properly designed formalism is a time and space saver, time for both the reader and the writer, and space in being more compact. We see intuitive solutions that are compact because they omit essential details or ignore corner cases. In the hands of a Dijkstra, formalism is a formidable tool, not a cross to bear. Formal methods may take longer to

learn, but once mastered they should pay off amply, much as algebra or calculus beats commonsense reasoning. I want to emphasize that not everything needs to be formal. In fact, gratuitous formalism adds to the burden in untangling all its details. A good engineer should know when she needs formal tools in order to be confident about her product and when formalism is simply a distraction because commonsense reasoning is equally convincing.

Simplicity and formalism, compactness and detailed explanation, and clarity and rigor are often conflicting goals. We cannot reduce the inherent complexity of an algorithm just by adopting one or another formalism. What we can do is apply the best that formal methods have to offer: apply known theories and develop specific theories, adopt appropriate notations, and structure the descriptions of algorithms.

2

Set Theory, Logic and Proofs

This chapter covers set theory and logic at an elementary level as background material. I assume that the reader has already been exposed to these topics at some level. Therefore, the treatment of this material is terse. The readers familiar with these topics may look through this chapter for the notations used for logical expressions and the style of rendering proofs, which I employ in the rest of the book; in particular, see Sections 2.6.2 (page 50), 2.7 (page 54) and 2.8 (page 59). The notation and style are useful in their own rights beyond their use in this book.

2.1 Set

2.1.1 Basic Concepts

A *set* is a collection of *distinct elements*. The set elements may be enumerated, as in {hamster, cat, dog}, denoting the set of pets in a home. In this book, set elements are mathematical objects, either *elementary* or *structured*, where structured objects consist of components that are themselves objects. A set is itself a structured object whose elements are its components. A set is either finite or infinite, by having a finite or infinite number of elements.

Any finite set can be defined by enumerating its elements, as in {3, 5, 7}. The order in which the elements are enumerated is irrelevant, so we have $\{3, 5\} = \{5, 3\}$ and $\{3, \{4, 5\}, 2\} = \{2, 3, \{5, 4\}\}$. Since a set has distinct elements, $\{3, 3, 5, 5\}$ is not a set. An *empty* set has no element; it is denoted by $\{\}$ or ϕ .

The types of elements in a set need not be identical. Consider, for example, the set $\{3, \text{true}, (5, 7), \{\{3, 5\}, 7\}\}$ which consists of an integer, a boolean, a tuple of integers and a set. Further, note that $3 \neq \{3\}$ and $\phi \neq \{\phi\}$.

The *cardinality* of a finite set S is the number of elements in S and is denoted by $|S|$. Thus, $|\{2\}| = 1$ and $|\phi| = 0$. Section 2.8.5 (page 64) shows that different infinite sets may have different cardinalities.

A related concept is *multiset* or *bag*. A bag is like a set except that the elements in it may be repeated, so for bags $\{3, 5\} \neq \{3, 3, 5, 5\}$.

Set membership Write $x \in S$ to denote that x is an element of set S and $y \notin S$ for y is *not* an element of S . Thus, $3 \in \{3, 5, 7\}$ and $\{3, 5\} \in \{3, \{3, 5\}\}$ whereas $4 \notin \{3, 5, 7\}$. An element of a set is also called a *member* of the set.

2.1.2 Mathematical Objects Used in this Book

A set element can be any kind of mathematical object. This book uses a small number of objects useful in basic computer science. The elementary objects used in this book are integers, booleans, characters (e.g., the Roman alphabet, the symbols on a keyboard or a specified set of symbols), integer and real numbers, natural numbers (i.e., non-negative integers), rational numbers (i.e., fractions of the form a/b for any integer a and non-zero integer b), and complex numbers.

Components of a structured object are either elementary or structured. Below, I list some of the structured objects used in this book.

2.1.2.1 Tuple

An n -tuple, where n is an integer and $n \geq 2$, consists of n components separated by commas and enclosed within parentheses. Thus, $(3, 5)$, $(3, 3)$ and $(true, \{3, 5\})$ are 2-tuples, or pairs, and $(3, 4, 5)$, $(5, 12, 13)$ are 3-tuples, or triples. Different components of a tuple need not be similar objects, so we may have a tuple for an individual whose first component is the name, which is a string, and the second component the age, an integer.

An n -tuple is different from a set because the order of elements in a tuple is relevant, that is, $(0, 1) \neq (1, 0)$, and the same element may appear several times in a tuple, as in $(1, 1)$, unlike in a set.

2.1.2.2 Sequence

A *sequence* is a finite or infinite list of elements that are ordered, so we have the first, second $\dots$ components of a non-empty sequence. Formally, a sequence is a totally ordered set; see Section 2.4.1 (page 32) for the definition of total order.

An enumerated (finite) sequence is enclosed within angled brackets, as in $\langle 2, 3, 5, 7, 11 \rangle$, or written without commas as in $\langle 2\ 3\ 5\ 7\ 11 \rangle$. The sequence without any element, the empty sequence, is written as $\langle \rangle$.

A *segment* of a sequence is a contiguous subset of its elements in order; so, $\langle 3, 5, 7 \rangle$ is a segment of $\langle 2, 3, 5, 7, 11 \rangle$. A *subsequence* x of sequence y consists of a set of elements of y in order, but the elements of x need not be contiguous in y . So, $\langle 2, 5, 11 \rangle$ is a subsequence of $\langle 2, 3, 5, 7, 11 \rangle$, though not a segment. And a segment is always a subsequence.

2.1.2.3 Character

A character, or a symbol, is written within a pair of single quotes, as in ‘m’, ‘3’ or ‘*’.

2.1.2.4 String

A string is a finite or infinite sequence of symbols, traditionally written within double quotes for finite strings instead of within angled brackets. String “male”, which is the same as $\langle \text{‘m’}, \text{‘a’}, \text{‘l’}, \text{‘e’} \rangle$, has four components, the first being the symbol ‘m’, the second ‘a’, and so forth. A segment of a string is called a *substring*.

2.1.2.5 Integer Interval

For integers s and t write $s..t$ to denote the sequence $\langle s, s+1, \dots, t-1 \rangle$, and call it an *integer interval*, or just *interval*. If $s \geq t$, then $s..t$ is empty. The length of the interval is $t - s$ for $s \leq t$ and 0 otherwise. Interval $s..t$ is a subinterval of $u..v$ if $u \leq s \leq t \leq v$.

The interval $s..t$ is written without brackets or parentheses, unlike the traditional notation for intervals in mathematics, so, $(s..t)$ is the same as $s..t$; here the parentheses serve their usual mathematical role of bounding the scope. I use $\{s..t\}$ for the set of integers in $s..t$.

Integer interval is a special case of *interval* defined in Section 2.4.2.3 (page 36).

2.1.2.6 Array, Matrix

I expect the reader to know arrays. Write $A[s..t]$ to denote the segment of an array A with indices ranging from s to $t - 1$. Declaring array A as $A[0..N]$ denotes that its indices range from 0 to $N - 1$, and $A[i..j]$ is a segment of A .

I expect the reader to know a *matrix* as a two-dimensional version of an array, but not much more than this. I supply the necessary material from linear algebra where it is needed. The notation for arrays extends to matrices; $M[1..M, 0..N]$ is a matrix whose rows are indexed from 1 through $M - 1$ and columns from 0 to $N - 1$, for example.

2.1.2.7 Tree

I give a brief description of trees here. Trees and graphs are covered in more detail in Chapter 6.

A *rooted tree*, or simply a *tree*, has a *root*. A root r has a set of *children*; each member of the set is a *child* of r , and r is the *parent* of all the children. The root and each child is called a *node*. Each child, in turn, is the root of a tree, called a *subtree* of r , and all its children are also nodes. Note that the set of children may be empty. There is an *edge* from a parent node to a child node. Figure 2.1 shows an example of a tree where a is the root and where the edges are drawn downward.

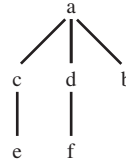


Figure 2.1 Rooted tree.

It follows that every node except the root has a unique parent, and the root has no parent. A node whose set of children is empty is a *leaf*. A smallest tree has only a root node that is also a leaf. A *forest* is a set of trees.

In this book, I show an edge as an arrow from a parent to a child; thus $u \rightarrow v$ is the edge from parent u to child v . A sequence of connected edges is a *path*. Figure 2.1 has several paths from root a ; I enumerate all paths from a to leaf nodes: $a \rightarrow c \rightarrow e$, $a \rightarrow d \rightarrow f$ and $a \rightarrow b$. The length of a path is the number of edges in it. I permit paths of length 0, from a node to itself. The *height* of a tree is the length of the longest path in it.

Node w is an *ancestor* of v and v a *descendant* of w iff there is a path from w to v . Thus, every node is an ancestor and descendant of itself because of paths of length 0, and *root* is the ancestor of every node. A node is reachable only from its ancestors. Every node in a tree is the root of a subtree that consists of the nodes reachable from it.

Binary tree An important class of trees is the *binary tree*, in which every node has at most two children. The children are usually identified as left and right children, the subtrees of which they are the root are the left and right subtrees, respectively. A notable fact is that a binary tree of height h has at most 2^h leaf nodes.

2.1.2.8 Function, Relation

Functions and relations are described in Sections 2.2 (page 17) and 2.3 (page 25), respectively. Each is an object in its own right, and it can be applied to construct other objects.

2.1.3 Set Comprehension

It is possible, in principle, to define any finite set through enumeration, though it may be impractical; consider enumerating all even numbers up to a million. And, an infinite set cannot be written down in its entirety. *Set comprehension* defines a set by describing a property that holds for all and only its elements. The property is given by a predicate so that an element is in the set if and only if it satisfies that predicate (see Section 2.6 (page 50) for a formal definition of predicate). A definition by comprehension takes the form $\{x \mid P(x)\}$, where x is a variable and P a

predicate that has x as a free variable (see Section 2.6.1 (page 50) for definition of free variable). The set consists of exactly the elements that satisfy P . The choice of a name for the variable is irrelevant; any name can be chosen, so the sets $\{x \mid P(x)\}$ and $\{y \mid P(y)\}$ are identical.

As an example of comprehension, the set consisting of all integers between 0 and 10 inclusive is $\{x \mid x \in \text{integer}, 0 \leq x \leq 10\}$, where *integer* denotes the set of integers. Observe that there are two predicates used in this definition though, formally, they are equivalent to a single predicate. I show in Section 2.5.1 (page 37) that multiple predicates may be encoded by a single predicate using the logical “and” operator.

The set of all positive even integers up to a million is $\{x \mid 0 < x \leq 10^6, \text{even}(x)\}$, where *even*(x) holds for all and only even integers x . This set could have been defined by enumeration, though the definition would have been unwieldy. The set of all even integers is $\{x \mid \text{even}(x)\}$, which cannot be given by enumeration. If the predicate in the comprehension is not true of any value, the corresponding set is empty, as in $\{x \mid \text{even}(x), \text{odd}(x)\}$, which is the set of integers that are both even and odd.

A generalization of comprehension Write $\{f(x) \mid P(x)\}$ to denote the set of all $f(x)$, for some function f , where each x satisfies predicate P . This generalization allows us to write $\{x^2 \mid x \in \text{integer}, x \geq 0\}$ for the set of squares of natural numbers. And, $\{(x, y) \mid x + y = 0\}$ is the set of all pairs whose elements sum to 0; the types of x and y have to be specified elsewhere. The set of all Pythagorean triples is given by $\{(x, y, z) \mid x^2 + y^2 = z^2\}$, assuming that x, y and z are understood to be positive integers.

The problem with naive comprehension Defining a set by $\{x \mid P(x)\}$, where P is an arbitrary predicate over x , is often called *naive comprehension*. This scheme was proposed in Frege, see [van Heijenoort \[2002\]](#). Bertrand Russell wrote a letter to Frege in which he showed that such a definition could lead to a logical impossibility. A modern form of Russell’s proof is given Section 2.8.4 (page 63).

It is safe to use comprehension if the domain of x is a previously defined set S . The comprehension is then expressed explicitly as $\{x \mid x \in S, P(x)\}$, which defines the set to consist of all elements of S that satisfy P . Henceforth, I assume that the sets of integers, natural numbers (non-negative integers), positive integers, negative integers, real numbers and rationals are predefined. Further, data structures over these objects, such as triples of positive integers, are also predefined. A set can be defined recursively where the definition uses the name of the set being defined (see Section 3.6.1, page 122).

Sequence comprehension A sequence is a set whose elements are totally ordered; see Section 2.4.1 (page 32) for definition of total order. So, a sequence can be defined by defining a set, possibly using set comprehension, and then prescribing an order over its elements. I describe sequence comprehension in Section 2.4.1.1 (page 33) after introducing the notion of total order.

2.1.4 Subset

Set S is a *subset* of set T , written as $S \subseteq T$, means that every element of S is an element of T . And S is a *proper subset* of T , written as $S \subset T$, means that $S \subseteq T$ and $S \neq T$. Thus, $\{3, 5\} \subseteq \{3, 5\}$ and $\{\{3, 5\}\} \subset \{\{3, 5\}, 7\}$. Note that $\{3, 5\}$ is not a subset of $\{\{3, 5\}, 7\}$ because the latter set does not contain either 3 or 5 as an element, though $\{3, 5\} \in \{\{3, 5\}, 7\}$.

Several properties of the subset relation are immediate. For an arbitrary set S , $S \subseteq S$ and $\emptyset \subseteq S$. For sets S and T if $S \subseteq T$ and $T \subseteq S$, then $S = T$. Also, if $S \subseteq T$ and $T \subseteq R$, for some set R , then $S \subseteq R$. The set of positive integers is a proper subset of the set of natural numbers, and the latter is a proper subset of the set of integers. The set of integers is a proper subset of the set of rationals, which is a proper subset of the set of real numbers.

For finite sets S and T where $S \subseteq T$, $|S| \leq |T|$, and if $S \subset T$, then $|S| < |T|$.

Powerset Powerset of set S is the set of all subsets of S . For $S = \{0, 1, 2\}$, the powerset is $\{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$.

The powerset of $\emptyset$ is $\{\emptyset\}$. Thus, $\emptyset$ is always a member of any powerset. Note that the cardinality of $\emptyset$ is 0 and of $\{\emptyset\}$ is 1. The cardinality of the powerset of a finite set S is $2^{|S|}$. This result can be proved formally using induction (see Chapter 3); below I give an informal proof.

Consider set S with n elements, $n \geq 1$. Number its elements 1 through n . Now, any n -bit string defines a unique subset S' of S : include element i in S' if and only if bit i of the string is 1. Note that the empty set is represented by the string of all zeroes and the entire set S by the string of all ones. Conversely, by a similar construction, every subset S' of S defines a unique n -bit string: bit i of the string is 1 iff $i \in S'$. Hence, the number of subsets of S equals the number of n -bit strings, that is, 2^n .

2.1.5 Operations on Sets

The following basic operators on sets are applied to two sets to yield one set: *union*, *intersection*, *Cartesian product* and *difference*. The *complement* operator yields a single set from a given set and an implicit universal set.

2.1.5.1 Binary Operators: Commutativity, Associativity, Distributivity

A binary operator $\star$ is *commutative* if for all x and y whenever $x \star y$ is defined it is equal to $y \star x$. So arithmetic $+$ and $\times$ are commutative because $x + y = y + x$ and $x \times y = y \times x$ for all operands x and y . The operator $\star$ is *associative* if for all x, y and z , whenever $(x \star y) \star z$ is defined, it is equal to $x \star (y \star z)$, and conversely; so arithmetic $+$ and $\times$ are associative as well. Consequently, for associative $\star$ the parentheses are redundant, just write $x \star y \star z$. For associative $\star$, the value of $x_0 \star x_1 \star \cdots \star x_n$ can be computed by applying $\star$ to the adjacent operands in any order (however, see Section 2.5.3.1 (page 42) for a major exception). For a commutative and associative operator $\star$, $x_0 \star x_1 \star \cdots \star x_n$ can be computed by applying the operator to pairs of operands in any order independent of adjacency.

Other two arithmetic operators, subtraction and division, are neither commutative nor associative. Operators min and max on numbers are commutative and associative. String concatenation and matrix multiplication are associative but not commutative. Averaging of two numbers, $(x + y)/2$ for operands x and y , is a commutative operation but not associative.

Operator $\otimes$ *distributes* over operator $\oplus$ means that $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ and $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$ for all x, y and z whenever one side of the identity is defined. For example, the product operation over numbers distributes over sum because $x \times (y + z) = (x \times y) + (x \times z)$ and $(x + y) \times z = (x \times z) + (y \times z)$. However, sum does not distribute over product.

2.1.5.2 Union

Union of sets S and T , written $S \cup T$, is a set that contains only and all elements of both S and T . Either or both sets may be infinite. Formally, $x \in S \cup T$ means $x \in S$, $x \in T$ or x is in both S and T . Thus, $\{0, 1\} \cup \{2\} = \{0, 1, 2\}$ and $\{0, 1\} \cup \{1, 2\} = \{0, 1, 2\}$. Observe that union is commutative and associative, and if S is a subset of T , then $S \cup T = T$. Therefore, $S \cup \phi = S$ and $S \cup S = S$.

The union operation does not allow cancellation like the addition operation on integers. For example, $a + b = c + b$ implies $a = c$, but $S \cup T = S' \cup T$ does not mean that $S = S'$, as in $\{2\} \cup \{3\} = \{2, 3\} \cup \{3\}$.

Union of an infinite number of sets puts the elements of all sets into a single set. More formally, for a family of sets F , $\bigcup_{S \in F} S = \{x \mid x \in S, S \in F\}$.

2.1.5.3 Intersection

Intersection of sets S and T , written $S \cap T$, is a set that contains the elements that are in both S and T . Either or both sets may be infinite. Formally, $x \in S \cap T$ means

$x \in S$ and $x \in T$. Thus, $\{0, 1\} \cap \{2\} = \{\}$ and $\{0, 1\} \cap \{1, 2\} = \{1\}$. Intersection is commutative and associative, and $S \subseteq S \cup T$, then $S \cap T = S$. Also, $S \cap T \subseteq S$ and $S \subseteq S \cup T$. Note that $S \cap \phi = \phi$ and $S \cap S = S$.

Sets S and T are *disjoint* if their intersection is empty; that is, they have no common element.

Intersection of an infinite number of sets puts all their common elements in a single set.

2.1.5.4 Complement

Given is a set S that is a subset of some implicit universal set. The complement of S , written $\bar{S}$, is the set of elements that are in the universal set and not in S . For example, let integers be the universal set, and *evens* the set of even integers and *odds* the odd integers; then $\overline{\text{evens}} = \text{odds}$ and $\overline{\text{odds}} = \text{evens}$. Complement of the set of negative integers is the set of naturals and the complement of the non-zero integers is $\{0\}$. Observe that $\bar{\bar{S}} = S$.

2.1.5.5 Cartesian Product

The Cartesian product of S and T , written as $S \times T$, is a set of pairs with the first element from S and second element from T . Either or both S and T may be infinite. Formally, $S \times T = \{(x, y) \mid x \in S, y \in T\}$. Thus, $\{0, 1\} \times \{1\} = \{(0, 1), (1, 1)\}$. The Cartesian product of a finite number of sets is defined similarly. For example, $S \times T \times U$ is a set of triples where the elements of a triple, in order, are from S , T and U . The cardinality of $S \times T$, for finite sets S and T , is $|S| \times |T|$. The Cartesian product of an infinite number of sets is beyond the scope of this book.

Cartesian product is neither commutative nor associative. In particular, $(S \times T) \times U \neq S \times (T \times U)$ because an element of the former is a pair whose first component is a pair from $S \times T$ whereas each element of the latter is a pair whose first component is a single element of S .

Example 2.1 I model a very small part of the state of a car that deals with its dome light. The dome light is either on or off at any moment. It is controlled by a switch, which could be in one of three positions: on, off or neutral, and also by the position of a door, which could be open or closed. Enumerate all possible states of (light, switch, door) as triples.

Observe that there are $2 \times 3 \times 2 = 12$ possible states because the light has two possible states (on, off), the switch has three (on, off, neutral), and the door has two (open, closed). The straightforward way is to enumerate the 12 triples. The simpler alternative is to note that the system state is in: $\{\text{on}, \text{off}\} \times \{\text{on}, \text{neutral}, \text{off}\} \times \{\text{open}, \text{closed}\}$. This defines the set of possibilities for each component of the car

separately; so, it accommodates design changes more readily. We can write the same set using comprehension, which aids our understanding:

$$\begin{aligned} & \{light \mid light \in \{on, off\}\} \\ & \times \{switch \mid switch \in \{on, neutral, off\}\} \\ & \times \{door \mid door \in \{open, closed\}\}. \end{aligned}$$

The design of the lighting system places certain constraints on these states. The light is on/off depending on the switch being on/off independent of the door position. If the switch is in the neutral position then, the light is on iff the door is open. Thus, a state such as (on, off, open) is invalid because the light is never on if the switch is off. Additional constraints may be added to define valid states.

2.1.5.6 Difference, Symmetric Difference

The difference $S - T$ of sets S and T is the subset of elements of S that are not in T , that is, $S - T = \{x \mid x \in S, x \notin T\}$. Thus, $\{2, 3\} - \{1, 2\} = \{3\}$ and $\{2, 3\} - \{1, 0\} = \{2, 3\}$. Given a universal set U of which both S and T are members, $S - T = S \cap \bar{T}$, where $\bar{T} = U - T$. The *symmetric difference* of S and T is $(S - T) \cup (T - S)$; this operator is not used in this book.

Set difference is neither commutative nor associative. For any S and T where $S \subseteq T$, $S - T = \phi$, and $S - \phi = S$ and $\phi - S = \phi$. Symmetric difference is commutative and associative.

2.1.5.7 Operations on Sequences

The operations on sets can be applied to sequences provided the order of the elements in the resulting sequence can be determined. For example, union and intersection over $\langle 2, 3, 4 \rangle$ and $\langle 4, 3, 2 \rangle$ are not defined because the elements occur in different order in the two sequences. A useful operation on finite sequence S is its reversal, $rev(S)$. For sequence S and set T intersection $S \cap T$ and difference $S - T$ are defined because the order of the elements in the resulting sequence can be determined.

2.1.6 Properties of Set Operations

The following laws are essential for proving identities on sets and their subsets. Below, U is a universal set such that every set under consideration is a subset of it.

2.1.6.1 Laws about Sets

(Identity Law) Union with ϕ or intersection with U yields the same set:
 $S \cup \phi = S$, $S \cap U = S$.

(Commutative Law) $\cup, \cap$ are commutative:

$$S \cup T = T \cup S, S \cap T = T \cap S.$$

(Associative Law) $\cup, \cap$ are associative:

$$(S \cup T) \cup R = S \cup (T \cup R), (S \cap T) \cap R = S \cap (T \cap R).$$

(Distributive Law) $\cap$ and $\cup$ distribute over each other:

$$S \cup (T \cap R) = (S \cup T) \cap (S \cup R), S \cap (T \cup R) = (S \cap T) \cup (S \cap R).$$

(Complement Law) $\cup$ and $\cap$ with complements yield U and ϕ , respectively:

$$S \cup \bar{S} = U, S \cap \bar{S} = \phi. \quad \blacksquare$$

These five laws about sets are fundamental; other laws can be derived from them. These include the Idempotent Laws: $S \cup S = S$ and $S \cap S = S$, Subsumption Laws: $S \cap \phi = \phi$ and $S \cup U = U$, and Absorption Laws: $S \cup (S \cap T) = S$ and $S \cap (S \cup T) = S$.

Observe that the laws always come in pairs and they are symmetric in $\cup$ and $\cap$, and U and ϕ . That is, interchanging $\cup$ and $\cap$, and U and ϕ in a law yields the other law in the pair (see Section 2.6.4, page 52).

2.1.6.2 Laws about Subsets

(Reflexivity) $S \subseteq S$.

(Antisymmetry) If $S \subseteq T$ and $T \subseteq S$, then $S = T$.

(Transitivity) If $S \subseteq T$ and $T \subseteq R$ then, $S \subseteq R$.

(Empty set and Universal set): $\phi \subseteq S$ and $S \subseteq U$.

(Inclusion):

$$(S \cap T) \subseteq S \text{ and } S \subseteq (S \cup T),$$

$$\text{If } S \subseteq T, \text{ then } S \cap T = S \text{ and } S \cup T = T.$$

2.1.6.3 Venn Diagram

Books on elementary set theory typically include pictorial representations of sets using a “Venn diagram”. A Venn diagram represents each set under consideration by a closed region. All elements of a set are taken to be points within the given region. If no relationship between sets A and B is known, then their regions intersect arbitrarily. If C is a subset of A , then C ’s region is completely within A ’s, and if two sets are disjoint, then the corresponding regions do not intersect at all. The intersection of two sets is the region common to both and their union is the region that includes exactly both regions, which need not be closed.

Figure 2.2 shows a Venn diagram in which each circular and elliptical region represents the set whose name appears within the region. No relationship between

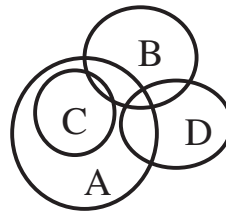


Figure 2.2 Venn diagram of sets A , B , C and D .

sets A and B is known, C is a subset of A but has no known relationship with B , and D has no known relationship with A or B though it is disjoint from C . Observe that D consists of four disjoint regions: $\bar{A} \cap \bar{B} \cap D$, $\bar{A} \cap B \cap D$, $A \cap B \cap D$ and $A \cap \bar{B} \cap D$. Some simple facts can be deduced from the picture, such as $\bar{A} \cap B$ is disjoint from C .

Venn diagrams are not particularly useful for reasoning about multiple sets or doing algebra on sets. In particular, there is no way to differentiate the case where A and B are known to have some common elements from the case when nothing at all is known about them. More importantly, the diagrams do not easily scale up to represent a large number of sets. So, I will not be using them in this book.

2.2 Function

2.2.1 Basic Concepts

A *function* is given by two non-empty sets, *domain* S and *codomain* T , and a *mapping* of each element of S to some element of T . Write $f: S \rightarrow T$ to denote that f is a function with domain S and codomain T , and say that f is *from* S *to* T . Either or both of S and T may be infinite. The mapping (often called a *graph* of the function) is a subset of $S \times T$ with the restriction that every element x of S appears in exactly one pair (x, y) . Then f maps x to y and $f(x)$ denotes y . In this notation x is the *argument* of f in $f(x)$.

Consider the function $intval: \{true, false\} \rightarrow \{0, 1\}$ that maps *true* to 1 and *false* to 0. The mapping in $intval$ is given formally by the set $\{(true, 1), (false, 0)\}$. Define function $intval'$ to be the same mapping as $intval$ except that the codomain of $intval'$ is $\{0, 1, 2\}$. Even though the two functions exhibit similar behavior when implemented on a computer, they are formally different functions because of different codomains.

The mapping between the domain and the codomain can be shown in a picture. For example, consider $f: \{0, 1, 2, 3, 4\} \rightarrow \{0, 1, 2, 3, 4\}$ where $f(x) = x^2 \bmod 5$.

See Figure 2.3 where an arrow from x in the domain to y in the codomain means that $f(x) = y$. There is exactly one outgoing arrow from each element of the domain to some element of the codomain (in this case the domain and codomain are equal). Some elements of the codomain may have no incoming arrows, such as those labeled 2 and 3, and some may have more than one, such as 1 and 4.



Figure 2.3 Graph of a Function.

Every element of the domain of a function is mapped to exactly one element of the codomain, but there is no similar requirement about the elements of the codomain; that is, there may be no element in the domain that maps to a specific element of the codomain, as is the case for 2 in the codomain of f in Figure 2.3.

Example 2.2 The familiar arithmetic operations $+$, $-$ and $\times$ are functions from pairs of reals, say (x, y) , to a real, $x + y$, $x - y$ and $x \times y$, respectively. The division function is similar except that its domain is $R \times (R - \{0\})$, where R is the set of reals, because the function value is not defined when the divisor is 0. The non-negative square root function applied to non-negative real x is written as $|\sqrt{x}|$, where $(|\sqrt{x}|)^2 = x$. The non-positive square root function is similarly defined.

Note on writing style It is sloppy to write “Let $f(x) = \sqrt{x}$ be a function”. First, the name of the function is f , and not $f(x)$, which has to be inferred from the context. Further, its domain and codomain are unspecified. It is possible that the domain is the set of non-negative reals and the codomain is the reals, but it is also possible that the domain is reals and the codomain the complex numbers. Also, $\sqrt{x}$ may be positive or negative, so the exact function value has to be specified. A more acceptable form of informal writing is “ f is a function from the non-negative reals to the non-negative reals where $f(x) = +\sqrt{x}$ ”.

2.2.1.1 Function Arity

The example functions that we have seen so far have just one argument. A function can have n arguments, $n \geq 0$, for any fixed integer n . The number of arguments n is called the *arity* of the function, and a function with n arguments is a n -ary function. A unary function has one argument, as in f where $f(x) = |\sqrt{x}|$. Familiar arithmetic operations, such as $+$ and $\times$, are 2-ary. Function *Pythagorean* has positive integer arguments x , y , and z , and its value is *true* iff $x^2 + y^2 = z^2$; so it is a 3-ary function. A related function *Pythagorean'* has a single argument, a *triple* of

positive integers (x, y, z) , and the same value as that of *Pythagorean*; so *Pythagorean'* is a unary function.

The arguments may be of different kinds, say integer and boolean for a function with two arguments. The domain of a function is the Cartesian product of the domains of the associated arguments.

A function with no argument, a 0-ary function, has a fixed value; so it is a constant. And every constant is a 0-ary function. Often, the domain of a constant function is omitted or left to the discretion of the reader. To be precise, however, the domain should be specified. The codomain is again left unspecified, though it must include the constant itself as a value.

2.2.1.2 Function Definition and Computability

The definition of *function*, as mapping from a domain to a codomain, is not “natural” at first sight. Any expression, say $x^2 - y + 3$ over integers, defines a function $f: \text{integer} \rightarrow \text{integer}$ and provides a procedure for computing its value for any given values of its arguments. For a long time, it was accepted that a function should be defined by a computational procedure, that is, prescribe a procedure to transform any argument value from the domain of the function to its value in the codomain. In the early part of the 20th century mathematicians realized that there are functions, now known as *uncomputable functions*, for which there is no computational procedure. So, the more general definition, which we currently adopt, was proposed. I do not discuss computability in this book.

2.2.1.3 A Taxonomy of Functions

A function is called *total* if every element of the domain maps to some element of the codomain; it is *partial* otherwise. The standard arithmetic functions $+$, $-$ and $\times$ over integers and real numbers are total. The division function, $/$, is also total over the domain $R \times (R - \{0\})$, where R is the set of reals, so the function is defined for all real divisors except 0. Any partial function can be converted to a total function by restricting its domain appropriately. I consider only total functions in this book.

An *injective* (or *one-to-one*) function maps every element of the domain to a distinct element of the codomain. A *surjective* (or *onto*) function maps some element of the domain to any given element of the codomain. A *bijective* function is both injective and surjective. Given a bijective function, there is a 1–1 correspondence between the elements of the domain and the codomain. Write *injection*, *surjection* or *bijection*, respectively, for a function that is injective, surjective or bijective. Given $f: S \rightarrow T$ for finite S and T , $|S| \leq |T|$ if f is injective, $|S| \geq |T|$ if surjective, and $|S| = |T|$ if bijective.

Consider $S = \{0, 1, 2\}$ and function $f: S \rightarrow S$ where $f(x) = (x + 1) \bmod 3$. Then f is bijective. Function $intval: \{true, false\} \rightarrow \{0, 1\}$, which was introduced earlier, maps *true* to 1 and *false* to 0, so it is bijective. Function $intval'$, which is the same mapping as $intval$ except that the codomain of $intval'$ is $\{0, 1, 2\}$, is only injective because no element in the domain maps to 2.

An important function is the *identity* function, $id_S: S \rightarrow S$, that maps every element of its domain to itself. The identity function on any domain is bijective.

2.2.1.4 Cantor–Schröder–Bernstein Theorem

This famous theorem says that if there are two injective functions $f: S \rightarrow T$ and $g: T \rightarrow S$, there is a bijective function between S and T . The result is not obvious and several earlier proof attempts were either faulty or relied on an axiom (axiom of choice) that is not really necessary. See Hinkis [2013] for a proof. A simple proof follows from the Knaster–Tarski theorem, Section 2.8.8 (page 70), but I do not describe it in this book.

2.2.2 Function Composition

Expression $f(g(x))$ denotes first applying g to some element x and then applying f to the result. This is meaningful only when $g(x)$ is in the domain of f . Typically, we assume that the domain of f is the codomain of g ; that is, $g: R \rightarrow S$ and $f: S \rightarrow T$. The composition of f and g in this form is a function, denoted by $f \circ g$, where $f \circ g: R \rightarrow T$, and $(f \circ g)(x) = f(g(x))$. A pictorial representation of function composition is given in Figure 2.4 where bold lines (with arrows) show functions f and g and thinner lines $f \circ g$.

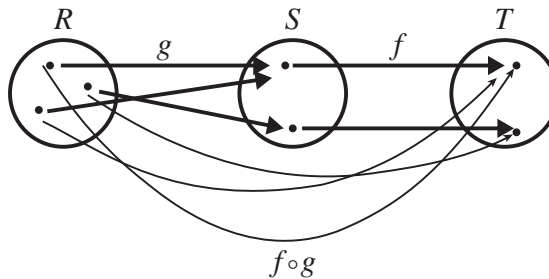


Figure 2.4 Schematic of $f \circ g: R \rightarrow T$, where $g: R \rightarrow S$ and $f: S \rightarrow T$.

A compiler for a programming language typically consists of multiple “passes”, where each pass is applied to the result of the previous pass. The first pass may do some lexical analysis to convert the program to one in an intermediate language, the next pass parses the program in the intermediate language and converts it to

a program in yet another language, and so on. So, each pass is a function and the compiler itself is a composition of these functions.

The composition of multiple functions, say f , g and h , written $f \circ (g \circ h)$, has the expected meaning provided the domain of a function in this sequence is the same as the codomain of the next function; so, f has the same domain as the codomain of $g \circ h$. Now, f , g and h can be combined to form $(f \circ g) \circ h$ under the same condition on domain and codomain, and it can be shown that $f \circ (g \circ h) = (f \circ g) \circ h$. That is, function composition is associative. The composed function acquires the domain of the last function and the codomain of the first. Function composition is not necessarily commutative.

2.2.3 Function Inverse

Let $f: S \rightarrow T$ and $g: T \rightarrow S$ be such that for any x and y if $f(x) = y$ then $g(y) = x$; that is, applying f and g in order starting with element x yields x . Then f and g are *inverses* of each other. The inverse of any function f , if it exists, is unique and denoted by f^{-1} .

As an example, let $S = \{0, 1\}$ and $T = \{true, false\}$. Let $f(0) = true$ and $f(1) = false$. Function f and its inverse g are shown pictorially in Figure 2.5.

Consider $S = \{0, 1, 2\}$ and $g: S \rightarrow S$ where $g(x) = (x + 1) \bmod 3$. Then $g^{-1}: S \rightarrow S$ where $g^{-1}(x) = (x - 1) \bmod 3$.

The reader may show that $f: S \rightarrow T$ has an inverse if and only if it is bijective. Further: (1) $f \circ id_S = f$ and $id_T \circ f = f$, (2) $f \circ f^{-1} = id_T$ and $f^{-1} \circ f = id_S$, (3) $(f^{-1})^{-1} = f$, and (4) $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$.

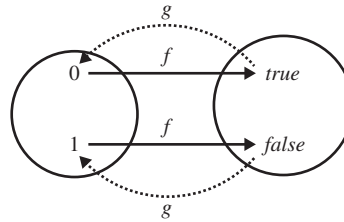


Figure 2.5 Pictorial depiction of function f and its inverse g .

Example 2.3 The position of a point in the two-dimensional plane can be given by a pair of reals, called its rectangular coordinates, or by another pair of reals, a length and measure of an angle, called its polar coordinates. Define function f to map rectangular coordinates (x, y) of a point to its polar coordinates (r, θ) , where $r = \sqrt{x^2 + y^2}$ and $\theta = \tan^{-1}(y/x)$. The inverse function g maps polar coordinates to rectangular coordinates: $x = r \cdot \cos(\theta)$ and $y = r \cdot \sin(\theta)$.

2.2.4 One-way Function

Call function f *one-way* if it is “easy” to compute $f(x)$, but “hard” to compute x given $f(x)$. Function f may or may not be bijective. The terms easy and hard are precisely defined using the theory of computational complexity, a subject I do not treat in this book. In informal terms, “easy” means that the function value can be computed within time that is a (small) polynomial of the size of its argument, whereas “hard” in this context means that any such method is almost certain to fail, for a randomly chosen x . Consequently, computing a y for which $f(y) = f(x)$ is almost certain to take a prohibitively long time.

One-way functions are important in many computer applications, particularly in cryptography. As a small example, consider a system that allows user-login with a name and a password. The system has to store the names and passwords to validate a login, but a system breach may expose these files to hackers. Almost all systems store the passwords by applying a one-way function f to each password; that is, instead of storing password p the system stores $f(p)$, called the *encrypted* password. During a login the user input of password x is converted to $f(x)$ and checked against the stored encrypted passwords. A system breach exposes the values of all encrypted passwords, from which the passwords themselves would be hard to reconstruct.

Consider the function over pairs of primes p and q whose value is $p \times q$. It is easy to compute the function value given p and q , but there is no known easy algorithm to compute p and q from $p \times q$ for arbitrary primes p and q . So, this function is *probably* hard, but it is as yet unknown if the decomposition of a product of two primes into its factors is hard.

2.2.5 Monotonic Function

A *monotonic* function’s values continue in the same direction, higher or lower, as its argument values are increased. For the moment, assume that argument and function values are either reals or integers. A *monotonically increasing* function f has a higher value when its argument’s value is increased; so, for x and x' in the domain of f where $x < x'$, $f(x) < f(x')$. A *monotonically decreasing* function has a lower value when its argument’s value is increased. *Monotonically ascending* and *descending* functions are similarly defined: given a monotonically ascending function f' , and x and x' in its domain where $x \leq x'$, conclude that $f(x) \leq f(x')$; similarly for monotonically descending functions. I develop a more elaborate theory of comparing values in the domain and codomain in Section 2.4 (page 32).

Monotonic functions are ubiquitous in computer science. Sum and product over positive reals are monotonic. Observe that sum over reals is monotonically increasing in both arguments, that is, if either argument is increased in value, the

function value increases. Product over positive reals is also monotonically increasing in both arguments. However, if y is negative, $x \times y$ decreases in value for increasing values of x . The reciprocal function x/y over positive reals x and y is monotonically increasing in x and decreasing in y for increasing values of y .

Monotonic sequence A sequence of values is *monotonically increasing*, or just *increasing*, if each element of the sequence is strictly smaller than the next element, if any. Monotonically decreasing, monotonically ascending and monotonically descending sequences are similarly defined.

2.2.6 Higher-order Function

A *higher-order* function has a function as an argument. A simple example is function *apply* that has two arguments, f and x , where $\text{apply}(f, x) = f(x)$. Then, $\text{apply}(\text{id}_R, 2) = 2$, $\text{apply}(\text{square}, 2) = 4$ and $\text{apply}(\text{cube}, 2) = 8$, where R is the set of reals and *square* and *cube* have their expected meanings. I treat higher-order functions in some detail in Section 7.2.9 (page 400).

2.2.7 Pigeonhole Principle

The pigeonhole principle is an outstandingly effective tool in combinatorial mathematics, based on an outstandingly simple observation about injective functions over finite domains. Suppose m pigeonholes are occupied by n pigeons, where pigeons outnumber the pigeonholes, that is, $n > m$. The principle asserts that some pigeonhole is occupied by more than one pigeon. The principle is obvious: if every pigeonhole is occupied by at most one pigeon, then the number of pigeons is at most m , contradicting that the pigeons outnumber the pigeonholes.

A more formal way to look at this problem (which is an overkill, but it provides a different kind of insight) is that pigeon-nesting is a function from the set of pigeons to the set of pigeonholes. If the function is injective, see Section 2.2.1.3 (page 19), there are at most as many pigeons as pigeonholes, if surjective there are at least as many pigeons as pigeonholes, and if bijective these numbers are equal.

There is a very useful generalization of this principle in [Dijkstra \[1991\]](#): the maximum number of pigeons in a hole is at least the average, that is, greater than or equal to $\lceil n/m \rceil$, and the minimum is at most the average, that is, less than or equal to $\lfloor n/m \rfloor$, irrespective of whether n exceeds m or not. Here $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ are the *ceiling* and *floor* functions, respectively. Again, this observation is obvious. A consequence is that in two-way voting with an odd number of voters, one contender is bound to win a majority of votes.

The principle is often used in connection with the problem of distributing m letters in n mailboxes (letters are pigeons and mailboxes pigeonholes), or coloring m balls using n colors (balls are pigeons and colors pigeonholes). In each case there

are at least $\lceil n/m \rceil$ letters/balls in the same mailbox/color. Let us see a few less trivial examples.

Given m natural numbers, at least two have the same value modulo t , where $0 < t < m$. To see this, let there be t different pigeonholes numbered 0 to $t - 1$. Treat each number k as a pigeon that is assigned to the pigeonhole with number $(k \bmod t)$. Since there are more pigeons than pigeonholes, at least two numbers occupy the same pigeonhole, that is, have the same value modulo t .

The population of the US exceeds 328 million, as of this writing. Show that over 890,000 people have the same birthday. Use 366 possible birthdays as pigeonholes, people as pigeons, and Dijkstra's formulation to deduce that the maximum number of people having the same birthday is at least $328 \times 10^6 / 366$, which is greater than 890,000.

A puzzle asks for placing five points within a unit square so that they are as far apart as possible, that is, the minimum distance between any pair of them is as large as possible. I show an upper bound on the minimum distance. Divide the square into four $1/2 \times 1/2$ squares. Since there are five points and four squares, at least two points occupy the same square. The maximum distance between any two points in a square is along the diagonal, and the length of the diagonal in a $1/2 \times 1/2$ square is $1/\sqrt{2}$. So, there is a pair of points that are no more than $1/\sqrt{2}$ apart.

The *handshake problem* is about a finite number of persons each of whom shakes some number of hands, possibly zero. Show that at least two persons shake the same number of hands. Suppose there are n persons. For each i , $0 \leq i < n$, create a pigeonhole representing i handshakes. Each person, a pigeon, is placed into the pigeonhole based on the number of hands she shakes. There are n pigeonholes, but not both of 0 and $n - 1$ are occupied because if a person shakes 0 hands no person can shake $n - 1$ hands, and conversely. So, the occupied number of pigeonholes is at most $n - 1$ and n pigeons occupy them. So, some hole is occupied by more than one pigeon, that is, there are two persons who shake the same number of hands, possibly 0.

2.2.7.1 Minimum Length of Monotone Subsequence

Consider a finite sequence of numbers $s = \langle s_0, s_1, \dots \rangle$. An ascending (descending) subsequence of s is a subsequence of s whose elements are non-decreasing (non-increasing). Call a subsequence monotone if it is either ascending or descending. Show that a sequence s whose length exceeds n^2 , for some given n , includes a monotone subsequence of length more than n .

The following proof is by Paul Erdős. Let a_i be the length of a longest ascending subsequence ending at s_i (that includes s_i) and d_i the length of a longest descending subsequence ending at s_i . I claim that for distinct i and j , $(a_i, d_i) \neq (a_j, d_j)$.

To see this, let $i < j$. Then if $s_i \leq s_j$, any ascending subsequence ending at s_i can be extended by appending s_j to it, so $a_i < a_j$. Similarly, if $s_i \geq s_j$, $d_i < d_j$.

Each a_i and d_i is at least 1 because there are ascending and descending subsequences that include only s_i . Suppose the longest monotone subsequence has length k . Create k^2 pigeonholes numbered (m, n) , $1 \leq m, n \leq k$. Put each s_i , as a pigeon, in the hole numbered (a_i, d_i) , if possible. From the argument in the last paragraph, two pigeons do not occupy the same pigeonhole. Since there are more than n^2 pigeons, they can be accommodated exclusively only if $k^2 > n^2$, or $k > n$.

2.3

2.3.1

Relation

Basic Concepts

A *relation* is a generalization of a function. Whereas the mapping of function $f: S \rightarrow T$ is a subset of $S \times T$ with the restriction that there is exactly one pair (x, y) in $S \times T$ for every x in S , relation $R: S \times T$ is just a subset of $S \times T$. The relation is sometimes written as $S R T$ or just $R \subseteq S \times T$.

Figure 2.6 shows an example of a relation, the courses being taken by some students. The figure shows the relation pictorially; an arrow drawn from x to y means that student x is taking course y , that is, (x, y) belongs to the relation.

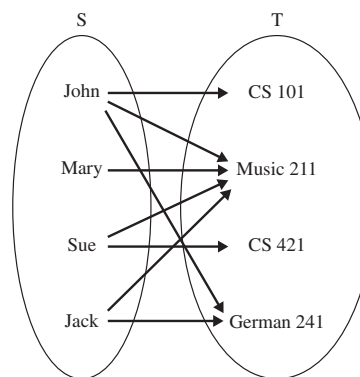


Figure 2.6 Pictorial depiction of a relation.

Relations are common in everyday life. There are “friends” and “knows” among a group of people, “cousin” in an extended family, and “belongs to” between cars and people. In the mathematical domain, we say 3 “divides” 6, where divides is between integers, 5 is “smaller than” 7, ab is a “substring” of abc , $\{3\}$ is a “subset” of $\{3, 5\}$ and node a is an “ancestor” of node x in a tree. A *vacuous* relation is empty,

that is, all pairs are unrelated, and the *identity* relation over $S \times S$ includes only pairs (x, x) for all x in S .

Relation R is called a *binary* relation when it is over two sets, S and T . Given any two items from S and T , either the relation holds between them (they are related) or does not hold (they are unrelated). Typically, binary relations are written in infix style, as in xRy when $(x, y) \in R$. Write $x \not R y$ if xRy does not hold. So, $3 < 5$ and $5 \not< 3$.

A multiary, or n -ary relation, $R: S_1 \times S_2 \times \dots \times S_n$, is defined over n sets, where $n \geq 2$. The relation is a subset of the Cartesian product of the sets in the given order. Such a relation can be written as a set of n -tuples. An example of a 3-ary (ternary) relation over integers x, y and positive integer n is given by the identity $x \equiv y \pmod n$; every triple (x, y, n) satisfying this identity belongs to the relation. A 4-ary relation is given by the identity $x^n + y^n = z^n$ over positive integers x, y, z and n . The famous Fermat's Last Theorem says that there is no quadruple (x, y, z, n) where $n > 2$.

A relation is a set; so, all operations on sets, such as union, intersection and difference, can be applied to relations. The *complement* of binary relation R , written $\bar{R}$, which is also $\not R$, is $\{z \mid z \notin R\}$. The *converse* of R , written R^{-1} , is the relation $\{(y, x) \mid (x, y) \in R\}$. And *composition* of binary relations R and R' , written $R; R'$, is the relation $\{(x, z) \mid (x, y) \in R, (y, z) \in R'\}$. Composition is an associative operation. I show a relation R , its complement, converse and composition with itself in Figure 2.7(a), (b), (c) and (d), respectively. It is a coincidence that $R^{-1} = R$; R . Notationally, composition uses the same symbol, “;”, used in programming for sequential composition¹.

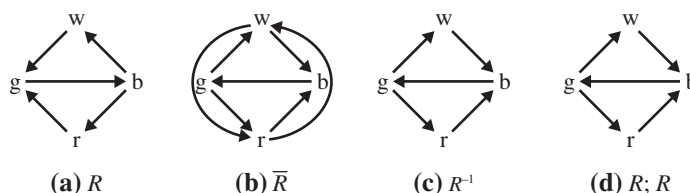


Figure 2.7 A relation, its complement, converse and composition.

2.3.2 Relational Databases

An important part of the computer industry is devoted to storing data of various kinds so that they can be manipulated in a variety of ways: adding new data or deleting existing data, combining existing data, and searching for data items that meet certain search criteria. It is now common to logically organize such data in the form of a *table*, where a table is merely a relation, a set of tuples.

1. A variety of notations have been used for relational composition. To signify the similarity with functional composition, $R \circ R'$ has been used in the past, but it is rarely used now. Sometimes simple juxtaposition, RR' , stands for $R; R'$.

I illustrate the idea with a small database of movies (see Table 2.1, page 27). I store the following information for each movie: its title, actor, director, genre and the year of release. I list only the two most prominent actors for a movie, and they have to appear in different tuples; so, each movie is represented by two tuples in the table. We can specify a search criterion such as, “find all movies released between 1980 and 2003 in which Will Smith was an actor and the genre is SciFi”. The result of this search is a table, as shown in Table 2.2 (page 28).

Table 2.1 A list of movies arranged in a table

Title	Actor	Director	Genre	Year
Jurassic World	Chris Pratt	Colin Trevorrow	SciFi	2015
Jurassic World	Bryce Dallas Howard	Colin Trevorrow	SciFi	2015
Men in Black	Tommy Lee Jones	Barry Sonnenfeld	Comedy	1997
Men in Black	Will Smith	Barry Sonnenfeld	Comedy	1997
Independence Day	Will Smith	Roland Emmerich	SciFi	1996
Independence Day	Bill Pullman	Roland Emmerich	SciFi	1996
Avengers: Endgame	Robert Downey Jr.	Anthony and Joe Russo	Fantasy	2019
Avengers: Endgame	Chris Evans	Anthony and Joe Russo	Fantasy	2019
Avatar	Sam Worthington	James Cameron	SciFi	2009
Avatar	Zoe Saldana	James Cameron	SciFi	2009
The Lion King	Chiwetel Ejiofor	Jon Favreau	Animation	2019
The Lion King	John Oliver	Jon Favreau	Animation	2019
Bad Boys II	Martin Lawrence	Michael Bay	Action	2003
Bad Boys II	Will Smith	Michael Bay	Action	2003
Ghostbusters	Bill Murray	Ivan Reitman	Comedy	1984
Ghostbusters	Dan Aykroyd	Ivan Reitman	Comedy	1984
Black Panther	Chadwick Boseman	Ryan Coogler	Drama	2018
Black Panther	Michael B. Jordan	Ryan Coogler	Drama	2018
Frozen	Kristen Bell	Chris Buck and Jennifer Lee	Animation	2013
Frozen	Idina Menzel	Chris Buck and Jennifer Lee	Animation	2013

Table 2.2 Result of selection on Table 2.1

Title	Actor	Director	Genre	Year
Men in Black	Will Smith	Barry Sonnenfeld	Comedy	1997
Independence Day	Will Smith	Roland Emmerich	SciFi	1996

I do not cover the topic of relational databases in this book; the topic is broad enough that many books have been written about it.

2.3.3 Important Attributes of Binary Relations

I list a few attributes of relations that occur commonly in practice. Two important classes of relations, *equivalence* (Section 2.3.4) and order relations (Section 2.4), that combine some of these attributes are treated in more detail in this chapter.

Reflexive Relation $R: S \times S$ is *reflexive* if xRx for all x in S . Examples of reflexive relation are equality of reals, “greater than or equal to” ($\geq$) over reals and “differs in no more than three positions” over strings of equal length. Relation “is a divisor of” over integers is not reflexive because division by 0 is not defined; “is a divisor of” over non-zero integers is reflexive. The following relations are not reflexive: $<$ over reals, “longer than” over finite strings and $\neq$ over any set on which equality is defined. In a non-mathematical domain, the relation “sister” is not reflexive. Relation R is *irreflexive* if for some x in its domain $x \not R x$.

Symmetric and asymmetric Relation $R: S \times S$ is *symmetric* if for all x and y in S , xRy holds whenever yRx holds. Equality over reals is symmetric whereas “greater than or equal to” is not because $5 \geq 3$ holds but $3 \geq 5$ does not hold. In a non-mathematical domain, the relation “sister” is not symmetric, as in “Alice is Bob’s sister” but “Bob is not Alice’s sister”. Relation R is *asymmetric* if for all x and y in S , whenever xRy holds yRx does not hold.

Antisymmetric Relation $R: S \times S$ is *antisymmetric* if for *distinct* elements x and y in S , both xRy and yRx do not hold. Equivalently, if both xRy and yRx hold, then $x = y$. Relation “greater than or equal to” over reals, subset relation over sets, substring relation over strings and subtree relation over trees are antisymmetric. The “disjoint” relation over finite sets—set S is disjoint from T if $S \cap T = \emptyset$ —is symmetric but not antisymmetric. Note that asymmetric and antisymmetric are different concepts.

Transitive Relation $R: S \times S$ is *transitive* if whenever xRy and yRz hold for any x, y and z in S then xRz . Relation “greater than or equal to” over reals, subset relation over sets, substring relation over strings and subtree relation over trees are transitive. Relation $\neq$ is not transitive (consider $3 \neq 5$ and $5 \neq 3$) and “differs in no

more than three positions” over strings of equal length is not transitive though it is reflexive and symmetric.

It is customary to write $a = b = c$ in mathematics with the intended meaning that $a = b$ and $b = c$. It then follows that $a = c$, using the transitivity of $=$. I use similar convention about other transitive operators, for example writing $x \geq y \geq z \geq u$, to denote that $x \geq z$, $x \geq u$ and $y \geq u$. Also, $\geq$ and $>$, as well as $\leq$ and $<$, can both be used in an expression, as in $x \geq y > z \geq u$. Also see Section 2.5.3.1 (page 42).

Example 2.4 A television program showed a baby solving the Rubik’s cube puzzle in less than a minute. The baby is normal, that is, devoid of any supernatural power, and the cube itself is genuine. The baby was given no instruction while solving the puzzle. Think about how it was done before reading the solution below.

The baby was given a cube that was already in its final configuration. The baby played with the cube and created arbitrary configurations. This process was filmed and played backwards in the program. This trick works only because the set of moves in Rubik’s cube puzzle is a symmetric relation; that is, if there is a move from configuration x to y , there is one from y to x ; so each move can be reversed. This trick will not work for a chess game because chess moves are not symmetric.

2.3.4 Equivalence Relation

Relation $R: S \times S$ that is reflexive, symmetric and transitive is called an *equivalence relation*. Relation R partitions S into disjoint subsets, called *equivalence classes*: x and y are in the same equivalence class if and only if xRy . I show that every element of S belongs to exactly one equivalence class. That is, if x and y are in an equivalence class and x and z are in an equivalence class, then x , y and z are all pairwise equivalent, and they are in the same equivalence class. Since xRy holds, yRx holds as well because R is symmetric. Using yRx , xRz and the transitivity of R , yRz holds.

Examples of equivalence relations in non-mathematical domains abound. People living in the same house belong to an equivalence class. So are citizens of the same country. Cars produced by a single manufacturer belong to the same equivalence class, called its “make”. Students who attend lectures from the same professor form an equivalence class. Postal service partitions mail for delivery into a hierarchy of classes. First, all mail to be delivered in a certain region (a city, zip code or post office area) are partitioned by region. Then, within each region mail are partitioned by addresses.

In the mathematical domain, consider the identity relation over any set S , which is an equivalence relation. Each equivalence class consists of just one element, so the number of equivalence classes is the size of S itself. For relation $(\text{mod } 3)$ over integers there are three equivalence classes: all integers that are evenly divisible by 3 (i.e., with remainder 0), those with remainder 1 and those with remainder 2.

The permutation relation over pairs of strings over a given alphabet is an equivalence relation; two strings belong to the same equivalence class iff they are permutations of each other.

An equivalence relation can be stored efficiently on a computer: with each element associate the equivalence class to which it belongs. For example, to determine if two cities have the same time zone, partition the set of cities into equivalence classes by time zone, then check if the given cities are in the same equivalence class.

An equivalence relation defined over an infinite set may have a finite or infinite number of equivalence classes. We have seen that the relation given by $(\text{mod } 3)$ over integers has three equivalence classes whereas the identity relation has an infinite number of classes. As another example, consider the set of finite binary strings, and define R such that xRy iff strings x and y have the same number of 1's; that is, $(01 R 010)$ holds but $(00 R 100)$ does not hold. It can be shown that R is an equivalence relation. Then all strings with n 1's form one equivalence class, for each natural number n .

2.3.4.1 Checking for Equivalence

A common problem in computing is to check if two items are equivalent according to some given equivalence relation. For example, in document processing it may be required to determine if two paragraphs are identical up to a permutation of their words, or more generally, if two lists have identical elements. The brute-force approach is to check each element of one list for its occurrence in the other and vice versa.

Unique representative Checking two items for equivalence is the same as determining if they belong to the same equivalence class. One strategy is to designate a specific member of each equivalence class as its *representative*. If there is an easy way to compute the representative given any member of a class, then equivalence check reduces to computing the representatives of the two items and checking if they are identical. In such a case, there is a function f such that $f(x)$ is the unique representative of the class to which x belongs, so $f(x) = f(y)$ iff x and y are equivalent. The effectiveness of this scheme depends on how easily $f(x)$ can be computed for any given x . See an extremely efficient algorithm, called *Union-Find*, in Section 6.7.2 (page 305) that can sometimes be used for this problem.

To check equivalence of two lists, let the representative be the sorted form of the list; so, they are equivalent iff their sorted versions are identical.

A degenerate example of unique representative is a bijective one-way function (see Section 2.2.4, page 22). Each element (e.g., password) is in an equivalence class by itself, and the unique representative is the encrypted form of the password.

2.3.4.2 Bloom Filter

Bloom filter in Bloom [1970], named after its inventor, is a technique for rapid probabilistic searching in a set. Let S be the set under consideration, initially empty, to which elements may be added from a universal set U in between searches. A negative search result, that is, the element is not in the set, is a correct result whereas a positive result may or may not be correct. The technique is useful in practice for those problems where the probability of a false positive can be made low.

A boolean array B of length m is used as a *proxy* for S . Choose a function $f: U \rightarrow \{0..m\}$, that is, f maps every possible value that can be added to S to some position in B . Initially, all values in B are *false*. Whenever element i is added to S , $B[f(i)]$ is set *true*. A search for j in S yields $B[f(j)]$. This is the correct output if j is in S (then the output is *true*), or no j' is in S where $f(j) = f(j')$ (then the output is *false*). However, for $f(j) = f(j')$, $j \notin S$ and $j' \in S$, the output would be *true*, which is incorrect. Summarizing, output *false* is always correct signifying that j is not in S whereas output *true* may be a false positive.

For any i , $B[i]$ is the unique representative of the equivalence class of all values j such that $f(j) = i$. Therefore, $B[i]$ being *true* merely signifies the presence of some member of the equivalence class in S , not any specific member.

This scheme can be improved considerably by choosing a number of hash functions $f_1, f_2, \dots, f_k$ instead of a single arbitrary function f . A hash function typically distributes the values in domain U uniformly over the interval $\{0..m\}$. With multiple hash functions whenever element i is added to S , $B[f_1(i)], B[f_2(i)], \dots, B[f_k(i)]$ are all set *true*. Search for j in S returns *true* iff $B[f_1(j)], B[f_2(j)], \dots, B[f_k(j)]$ are all *true*. If any one of these values is *false*, j is not in S . Use of multiple hash functions, say f_1 and f_2 , creates equivalence classes by taking the intersection of those corresponding to f_1 and f_2 , that is, j and j' are in the same equivalence class iff $f_1(j) = f_1(j')$ and $f_2(j) = f_2(j')$. A false positive in the search for j results if $j \notin S$ and $j' \in S$.

Ideally, the hash functions should be independent, but designing independent hash functions for large k is a difficult task. A number of useful heuristics, using variants of double hashing, are often employed.

2.3.5 Closure of Relation

For binary relation A over set S , its reflexive/symmetric/transitive closure is the smallest relation that includes A and is reflexive/symmetric/transitive.² Construct the reflexive closure of A by adding all pairs (x, x) , where x is in S , to A . And, construct symmetric closure of A by adding (x, y) to A for every (y, x) in A .

2. The name A used in this section for a relation is different from the name R used in the rest of this section. A is used to match the symbol for adjacency matrix used in graph theory, Chapter 6, where transitive closure is an important concept.

The transitive closure A^+ is more challenging. If $x_0 A x_1, x_1 A x_2, \dots, x_{n-1} A x_n$, then $x_0 A^+ x_n$. Transitive closure is an important concept. I define the problem for graphs, and more generally over semirings, in Section 6.8 (page 317). An efficient algorithm for its computation appears in Section 6.8.2 (page 319).

The reflexive, symmetric and transitive closure of A includes A , and is reflexive, symmetric and transitive. So, it is an equivalence relation.

Transitive closure of a graph Transitive closure has an interpretation in terms of graphs (see Section 6.8, page 317). Let $x A y$ denote that there is an edge from node x to node y and $x A^+ y$ denote that there is a finite path from x to y with at least one edge. The transitive closure A^+ then denotes the existence of paths with one or more edges, and the reflexive and transitive closure A^* denotes existence of paths with zero or more edges.

2.4 Order Relations: Total and Partial

I consider binary relations that impose order over the elements of sets. The familiar relation of “less than or equal to” over integers imposes an order over every pair of integers; so it is called a *total order*. A generalization of total order is *partial order* that orders some pairs of elements but not necessarily all pairs. Essential to a deeper understanding of induction is *well-founded* order, which is treated in Chapter 3.

2.4.1 Total Order

A *total order* orders every pair of elements of a set in a consistent fashion so that its elements can be arranged in a sequence where smaller elements precede the larger elements. For example, $\leq$ over integers induces the sequence $\langle \dots -2 \leq -1 \leq 0 \leq 1 \leq 2 \dots \rangle$.

Formally, binary relation R over set S is a *total order* if R is reflexive, antisymmetric and transitive, and for any two elements x and y of S either xRy or yRx . A total order is also called *linear order*, and set S is said to be totally, or linearly, ordered with respect to R . The set is also called a *chain* because its elements may be arranged in the form of a chain (i.e., a sequence) where xRy holds among adjacent elements.

A *strict order* is like a total order except that it is irreflexive. An example is $<$ over integers. Formally, strict order R is irreflexive, antisymmetric, transitive and for any two *distinct* elements x and y of S either xRy or yRx . There is a strict order corresponding to a total order and conversely. The strict order $<$ over integers corresponds to total order $\leq$. For total order R , the corresponding strict order R' is given by $xR'y$ if and only if xRy and $x \neq y$. Conversely, for strict order R' the corresponding total order R is given by: xRy if and only if $xR'y$ or $x = y$.

2.4.1.1 Sequence Comprehension Using a Total Order

A sequence is a totally ordered set. A sequence can be defined using comprehension to define a set and then prescribing a total order over it. I show a possible notation below where the order is implicit in the definition, and i ranges over integers, but I do not use this notation in the rest of the book.

1. $\langle i \mid 0 \leq i < 10 \rangle$ is the sequence 0..10.
2. $\langle i \mid 0 \leq i < 10, \text{even}(i) \rangle$ is the sequence of even natural numbers below 10.
3. $\langle i \mid 1 \leq i \rangle$ is the sequence of positive integers.

2.4.1.2 Lexicographic Order

Lexicographic order (also known as “dictionary order” or “alphabetic order”) is, informally, the order of words in a dictionary, and it is a total order. Consider a dictionary consisting of words of length 2 over the Roman alphabet. We order the words as in “ab” < “xy” and “ab” < “ac”. That is, given two words pq and rs , compare p and r first; if one is smaller, then the corresponding word is smaller, but if they are equal compare their second symbols to order them.

Formally, given strict total orders $R: S \times S$ and $R': T \times T$, where x and y are in S , and u and v are in T , define strict order $<$ over tuples by $(x, u) < (y, v)$ if either xRy or $(x = y \text{ and } uR'v)$. As usual, $(x, u) \leq (y, v)$ if $(x, u) < (y, v)$ or $(x, u) = (y, v)$. The reader should show that $\leq$ is a total order. I prove a more general result in Section 3.5.1.2 (page 117).

This definition is easily extended to compare n -tuples by comparing the elements of the tuples successively from left to right until one tuple entry is found to be smaller than the corresponding entry in the other tuple (if no such pair of entries is found, the tuples are identical). Positive integers of equal length, say 3461 and 3458, are compared by comparing their digits successively from left to right until 5 is found to be smaller than 6, so $3458 < 3461$. For comparing positive integers of unequal lengths, such as 23 against 121, the shorter one is padded with zeroes at the left end to make their lengths equal, so $023 < 121$. For comparisons of words in a dictionary, the shorter word is padded with blanks (white space) at the right end, and blank is taken to be smaller than every other symbol, so that “the” < “then”.³

2.4.1.3 An Example: Least-significant Digit Sort

In the early days, *least-significant digit sorting* was used to sort a deck of punched cards in certain columns, say from column 2 through 4. Sort first on column 4, then

3. This remark is true of trailing blanks, but customarily properly embedded blanks are ignored in dictionaries. However, in the older days, when we had telephone directories, embedded white spaces were low in the “directory order”. I am grateful to Doug McIlroy for this observation.

3, then 2, each time running the deck through a card-reading machine, called an accounting machine.

Consider the general case where the elements to be sorted are n -tuples, and each tuple entry is from a totally ordered set. The goal is to sort the entries lexicographically. A special case is to sort a table of numbers of equal length where the tuple entries are digits. I describe sorting for this special case; it is easily modified to apply to the general case.

A *pass* sorts the numbers according to their digits in a specific position. The sorting method applies passes from the least-significant digit position to the most significant with the following restriction: maintain the relative order of the numbers in a pass if their corresponding digits are identical. Thus, to sort the list $\langle 423, 471, 273 \rangle$ in ascending order, sort on the last digit to obtain $\langle 471, 423, 273 \rangle$ where the relative order of the last two numbers are preserved because their last digits are identical. The next pass, sorting on the second digit, yields $\langle 423, 471, 273 \rangle$ and the final pass $\langle 273, 423, 471 \rangle$.

The correctness argument is simple. Write x_i for the digit at position i of x . For any two numbers x and y in the list where $x < y$, there is some position j where $x_j < y_j$ and $x_i = y_i$ in all preceding positions i . Then after pass j , x will precede y in the list, and they will retain their relative orders in subsequent passes because they have identical digits preceding position j . This ensures that x will precede y in the final list.

2.4.2 Partial Order

Binary relation $\leq$ over set S is a *partial order* if $\leq$ is reflexive, antisymmetric and transitive; then S is *partially ordered with respect to* $\leq$. Unlike a total order, there is no requirement that every pair of elements be ordered. So, every total order, trivially, is a partial order, but not conversely. Henceforth, given $x \leq y$ in a partially ordered set say x is “less than or equal to” y or y is “greater than or equal to” x .

Partial orders are ubiquitous. The relation “divides” over integers, “subset” relation over sets and “ancestor” relation in a tree are some examples. School children are taught that the primary colors are red, blue and green. Mixture of red and blue produces magenta, red and green yields yellow, green and blue gives cyan, and the combination of all three colors gives white. Call color x higher than y if x can be obtained by mixing y with some other colors. Denoting red, blue and green by a , b and c , respectively, Figure 2.8 shows the partial order over the colors. The empty set denotes the absence of all primary colors, that is, black.

You can show that the union and intersection of two partial orders is a partial order.

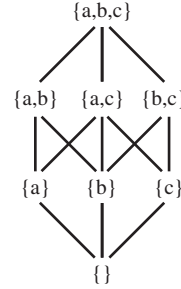


Figure 2.8 The subset order is a partial order.

2.4.2.1 Topological Order

A topological order $\leq$ is a total order for a given partial order $\preceq$ so that if $x \preceq y$ then $x \leq y$. I prove the existence of topological order for partial orders over finite, and certain classes of infinite, sets in Section 3.4.7 (page 111).

Given $x \preceq y$, imagine that x is a point at a lower height than y in a terrain. Then topological order arranges the points by their heights, from lower to higher heights.

As an example of topological order, consider the “divides” relation over positive integers, which is a partial order. A topological order is $\leq$, the standard order over integers, because if x divides y then $x \leq y$. For the subsets of $\{a, b, c\}$ shown in Figure 2.8, a possible topological order is:

$$\{\} < \{a\} < \{b\} < \{c\} < \{a, b\} < \{b, c\} < \{a, c\} < \{a, b, c\}.$$

There may be several topological orders for a given partial order. Another possible topological order for Figure 2.8 is:

$$\{\} < \{b\} < \{c\} < \{a\} < \{b, c\} < \{a, b\} < \{a, c\} < \{a, b, c\}.$$

2.4.2.2 Least Upper Bound, Greatest Lower Bound

Consider any set S , finite or infinite, with partial order $\leq$. Element z of S is an *upper bound* of the elements x and y if $x \leq z$ and $y \leq z$. And z is the *least upper bound*, abbreviated as *lub*, if for any other upper bound t of x and y , $z \leq t$. Least upper bound z of set T is similarly defined: (1) z is an upper bound, that is, $x \leq z$ for all x in T , and (2) for any other upper bound t of T , $z \leq t$. Write $x \uparrow y$ for the least upper bound of two elements x and y , if it exists, and $\text{lub}(T)$, explicitly, for the least upper bound of a set of elements T .

The least upper bound may not exist because there may be no upper bound or there may be several upper bounds though none of them is the least. The least upper bound, if it exists, is unique because for any two least upper bounds, z_1 and z_2 , $z_1 \leq z_2$ and $z_2 \leq z_1$, so $z_1 = z_2$ using the antisymmetry of $\leq$.

Consider a few examples of the least upper bound, where $\leq$ is some partial order and $\uparrow$ the corresponding least upper bound. Over the set of real numbers, let $x \leq y$ have its standard meaning; then $x \uparrow y$ is the maximum of x and y . However, the set of real numbers has no lub. Suppose $x \leq y$ means that y is an ancestor of x in a finite tree (assume x is its own ancestor); then $x \uparrow y$ is the lowest common ancestor of x and y , and the root is the lub of the entire tree. Over positive integers if $x \leq y$ means that x divides y , $x \uparrow y$ is the least common multiple of x and y .

The *greatest lower bound*, glb, is defined similarly. Element z of S is a *lower bound* of the elements x and y if $z \leq x$ and $z \leq y$. And z is the greatest lower bound if for any other lower bound t of x and y , $t \leq z$. The greatest lower bound, if it exists, is unique. Write $x \downarrow y$ for the glb of x and y , and $glb(T)$ for a set T . As an example, let $x \leq y$ mean that x divides y over positive integers; then $x \downarrow y$ is the greatest common divisor, gcd, of x and y .

2.4.2.3 Interval

Let a and b be elements of a partially ordered set $(S, \leq)$ where $a \leq b$. The *interval* between a and b is the partially ordered set that includes the elements that lie between a and b , where either, both or none of the end points a and b may be included in the set: $[a, b] = \{x \mid a \leq x \leq b\}$ is the *closed* interval that includes both end points, $(a, b) = \{x \mid a < x < b\}$ is the *open* interval that includes neither end point, and $[a, b) = \{x \mid a \leq x < b\}$ and $(a, b] = \{x \mid a < x \leq b\}$ are half-open (or half-closed) intervals that include just one end point. The partial order in the interval is the same as $\leq$, restricted to the elements of the interval.

A *real interval* is the set of real numbers between the two specified end points ordered in the standard manner. We defined the integer interval $s..t$ as the half-open interval $[s, t)$ (see Section 2.1.2.5, page 9).

2.4.2.4 Lattice

A *lattice* is a partially ordered set in which every pair of elements has a least upper bound and a greatest lower bound. It can be shown that a finite lattice has a unique least upper bound and a greatest lower bound. A complete lattice is defined in Section 2.8.8 (page 70) in connection with the proof of the Knaster–Tarski theorem.

We do not study lattices in any detail in this book except to note a very useful application of finite lattices in computer security. Any organization contains *objects*, such as files, directories and databases. Each object has a certain level of security associated with it, called its *security class*. Define a partial order $\leq$ over objects so that $a \leq b$ denotes that object b is at the same or a higher security class. In the following discussion, I do not distinguish between security classes and the objects they denote.

Typically, some objects are accessible to everyone; they are public knowledge so they have the lowest security class. If information from any two objects a and b are combined to form object c , then $a \leq c$ and $b \leq c$, that is information can flow only from a lower security class to a higher security class. It can be shown that any finite set of objects under these assumptions form a lattice. Figure 2.8 is a lattice on security classes a , b and c .

In any organization, each individual has access to only some subset of objects. Anyone having access to object b also has access to a if $a \leq b$. The individuals are also partially ordered according to their access privileges. The individuals need not form a lattice because there may be several individuals, with the topmost security clearance, who have access to all the objects; so, there is no unique least upper bound. Similarly, there may be no greatest lower bound.

2.4.2.5 Monotonic Function over Partially Ordered Sets

I generalize the notion of a monotonic function defined in Section 2.2.5 (page 22). Let D and R be partially ordered sets under relations $\leq_D$ and $\leq_R$, respectively. Function $f: D \rightarrow R$ is *monotonic* with respect to the given partial orders if for any x and y in D whenever $x \leq_D y$ then $f(x) \leq_R f(y)$.

2.5 Propositional Logic

I assume that the reader has had some exposure to formal logic. What follows is a quick summary of the main ideas of propositional and first-order logic. For a thorough treatment, see a book such as [Fitting \[1990\]](#), [Gries and Schneider \[1994\]](#) or [Mendelson \[1987\]](#). Readers familiar with the basic concepts should skim this section for notations that we employ in the rest of the book.

2.5.1 Basic Concepts

Consider the arithmetic expression $2 \times x - (3 + y)/x$. The expression includes constants 2 and 3, operations of addition, subtraction, multiplication and division, and variables x and y . Given real numbers (or complex numbers) as values of the variables, we can compute the expression value. We may calculate all pairs of values for x and y for which the equation $2 \times x - (3 + y)/x = 0$ is satisfied (if there is no solution to the equation, the associated set of pairs is empty).

A *boolean expression* is analogous. It has constants, boolean operators (sometimes called *boolean connectives*) and variables. There are just two constants, *true* and *false*, sometimes written T and F , respectively. We see only the following boolean operators in this book: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implication), $\Leftarrow$ (follows from), and $\equiv$ (equivalence). The value of a boolean variable is either *true* or

false, so the value of a boolean expression is also *true* or *false*. Examples of boolean expressions over variables p , q and r are:

$$p \vee (\neg q \wedge r), \quad (p \vee q) \wedge (p \vee r), \quad (p \vee (q \wedge \neg r)) \equiv (p \vee q) \wedge (p \vee \neg r).$$

Boolean expression, predicate, proposition I use the terms *boolean expression*, *predicate*, and *proposition* interchangeably in this book. Many authors reserve “proposition” for a boolean expression without quantifiers.

Informal meanings of boolean operators The boolean operators have the following informal meanings: $\neg p$ is *true* iff p is *false*, $p \wedge q$ is *true* iff both p and q are *true*, $p \vee q$ is *true* iff either or both of p and q are *true*, $p \Rightarrow q$ is *true* iff either or both of $\neg p$ and q are *true*, $p \Leftarrow q$ is *true* iff either or both of p and $\neg q$ are *true*, and $p \equiv q$ is *true* iff p and q have the same value. The equality relation, $=$, is also a boolean operator. It has the same meaning as $\equiv$ when applied to boolean operands, but $=$ is applicable over other types of operands.

Sentences such as “ p is necessary for q ” and “ p is sufficient for q ” mean, respectively, that $q \Rightarrow p$ and $p \Rightarrow q$. And “ p is necessary and sufficient for q ” means $p \equiv q$. The sentence “ p implies q ” can be written as either $p \Rightarrow q$ or $q \Leftarrow p$, and “ p follows from q ” can be written as either $p \Leftarrow q$ or $q \Rightarrow p$.

Conjunct, disjunct, literal In a boolean expression whose subexpressions are combined using $\wedge$, the subexpressions are called its *conjuncts*. Analogously, the subexpressions that are combined using $\vee$ are called the *disjuncts* of the expression. A variable and its negation, p and $\neg p$, are both called *literals*.

Truth table Each operand in a boolean expression has two possible values. Therefore, an expression that has n operands has 2^n possible combination of operand values. An expression’s value can be explicitly enumerated for each combination of operand values. Such an enumeration is called a *truth table*. In particular, the meanings of the boolean operators can be given formally by a truth table, as shown in Table 2.3.

Table 2.3 Definitions of propositional operators

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftarrow q$	$p \equiv q$	$(\neg p) \vee q$
<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>
<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>
<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>
<i>T</i>	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>

Table 2.3 also shows the value of $(\neg p) \vee q$ in the last column. We see that $p \Rightarrow q$ is the same as $(\neg p) \vee q$. Such facts can be deduced more readily using the laws of propositional logic given in Section 2.5.2 (page 40).

Precedences of operators Consider the expression $((\neg p) \wedge (q \wedge (\neg r))) \vee ((\neg q) \vee r)$. Every operand is within parentheses. The order in which the operators are to be applied to their operands is clear: first compute $((\neg p) \wedge (q \wedge (\neg r)))$ and $((\neg q) \vee r)$. Then apply the operator $\vee$ to the two computed values to obtain the result. The expression written in this form is said to be *fully parenthesized*. A fully parenthesized expression, however, is difficult to read and comprehend; consider an analogous arithmetic expression, $((-a) + (b + (-c))) \times ((-b) \times c)$, to appreciate the difficulty.

Arithmetic operators have *precedences* or *binding power*, such as $\times$ has higher precedence than $+$, to avoid fully parenthesizing an expression. We assign precedences to boolean operators for the same reason. The boolean operators in the decreasing order of precedences are given below where operators of equal precedence are within parentheses.

$$\neg, (\wedge, \vee), (\Rightarrow, \Leftarrow), \equiv, =.$$

Non-boolean operators have higher precedence than the boolean ones in a boolean expression. So, $x > y \Rightarrow x + 1 > y + 1$ means $(x > y) \Rightarrow ((x + 1) > (y + 1))$.

Note on operator precedence Observe that $\wedge$ and $\vee$ have the same precedence, so we use parentheses whenever there is a possibility of ambiguity. For example, instead of writing $p \wedge q \vee r$ we write either $(p \wedge q) \vee r$ or $p \wedge (q \vee r)$.

Operators $=$ and $\equiv$ have different precedences though they have the same meaning when applied to boolean operands. Thus, $p \equiv q = q \equiv p$ is the same as $(p \equiv q) = (q \equiv p)$.

To aid the reader in parsing a boolean expression visually, I often put extra white space around operators of lower precedences, as in $p \Rightarrow q \equiv q \Leftarrow p$. However, white spaces do not affect the precedences; the expression $p = q \wedge r = s$ does not denote $(p = q) \wedge (r = s)$, instead it is $p = (q \wedge r) = s$.

It is tempting to treat *true* and *false* as the numbers 1 and 0, and $\vee$ and $\wedge$ as the sum and product operators, respectively. The boolean operators do share a number of properties with their numerical counterparts, such as commutativity and associativity. However, $\vee$ distributes over $\wedge$ whereas sum does not distribute over product, that is, $p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$ holds whereas $x + (y \times z) = (x + y) \times (x + z)$ does not hold. The analogy with numerical operators, therefore, is misleading. The analogy has the unfortunate consequence that $\wedge$ has been given higher precedence than $\vee$ in many programming languages mimicking the higher precedence of product over sum.

An alternative to set definition by comprehension A set that is defined using comprehension, as in $S = \{x \mid P(x)\}$, can also be defined by $x \in S \equiv P(x)$, that is x is in S iff $P(x)$ holds. The alternative form is often useful for formal manipulations of predicates in constructing proofs.

Origin of boolean algebra Boolean algebra is named after George Boole. Boole embedded propositional logic within arithmetic and did not introduce special symbols for $\wedge$ and $\vee$, treating: $true = 1, false = 0, p \wedge q = p \times q$, and $p \vee q = p + q - p \times q$. He also noted that these are the laws of probability restricted to probability values of zero and one, where $\wedge$ is the probability of joint occurrence and $\vee$ the probability of either occurrence.

2.5.2 Laws of Propositional Logic

A truth table can be used to define each boolean operator. Defining the boolean operators using truth tables is similar to defining sum and product operators over integers using addition and multiplication tables. They are useful for beginners in calculating values of given boolean expressions, but far less so for algebraic manipulations. A number of laws—similar to commutativity of multiplication or distributivity of multiplication over addition—are essential. I list some of the laws below, grouping them under common headings to aid in memorization.

Below, p, q and r are boolean variables. I omit the laws about $\Leftarrow$ since it is rarely used in this book. All its laws can be derived using the identity $(p \Leftarrow q) = (q \Rightarrow p)$.

- (Commutativity) $\wedge, \vee, \equiv$ are commutative:

$$p \wedge q = q \wedge p$$

$$p \vee q = q \vee p$$

$$p \equiv q = q \equiv p$$
- (Associativity) $\wedge, \vee, \equiv$ are associative:

$$(p \wedge q) \wedge r = p \wedge (q \wedge r)$$

$$(p \vee q) \vee r = p \vee (q \vee r)$$

$$(p \equiv q) \equiv r = p \equiv (q \equiv r)$$
- (Idempotence) $\wedge, \vee$ are idempotent:

$$p \wedge p = p$$

$$p \vee p = p$$
- (Distributivity) $\wedge, \vee$ distribute over each other:

$$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$$

$$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$$

- (Absorption)

$$p \wedge (p \vee q) = p$$

$$p \vee (p \wedge q) = p$$

- (Expansion)

$$(p \wedge q) \vee (p \wedge \neg q) = p$$

$$(p \vee q) \wedge (p \vee \neg q) = p$$

- (Laws with Constants)

$$\text{Unit of } \wedge: p \wedge \text{true} = p \quad \text{Annihilator of } \wedge: p \wedge \text{false} = \text{false}$$

$$\text{Unit of } \vee: p \vee \text{false} = p \quad \text{Annihilator of } \vee: p \vee \text{true} = \text{true}$$

$$\text{Excluded middle: } p \vee \neg p = \text{true} \quad \text{Contradiction: } p \wedge \neg p = \text{false}$$

$$p \equiv p = \text{true} \quad p \equiv \neg p = \text{false}$$

$$\text{true} \Rightarrow p = p \quad \text{false} \Rightarrow p = \text{true}$$

$$p \Rightarrow \text{true} = \text{true} \quad p \Rightarrow \text{false} = \neg p$$

$$p \Rightarrow p = \text{true} \quad p \Rightarrow \neg p = \neg p$$

- (Double Negation)

$$\neg \neg p = p$$

- (De Morgan's Laws)

$$\neg(p \wedge q) = (\neg p \vee \neg q)$$

$$\neg(p \vee q) = (\neg p \wedge \neg q)$$

- (Implication operator)

$$(p \Rightarrow q) = (\neg p \vee q)$$

$$(p \Rightarrow q) = (\neg q \Rightarrow \neg p)$$

$$\text{(Transitivity): } ((p \Rightarrow q) \wedge (q \Rightarrow r)) \Rightarrow (p \Rightarrow r).$$

$$\text{(Monotonicity):}$$

$$((p \Rightarrow q) \wedge (p' \Rightarrow q')) \Rightarrow ((p \wedge p') \Rightarrow (q \wedge q'))$$

$$((p \Rightarrow q) \wedge (p' \Rightarrow q')) \Rightarrow ((p \vee p') \Rightarrow (q \vee q'))$$

- (Equivalence operator)

$$(p \equiv q) = (\neg p \equiv \neg q)$$

$$(p \equiv q) = (p \wedge q) \vee (\neg p \wedge \neg q)$$

$$(p \equiv q) = (p \Rightarrow q) \wedge (q \Rightarrow p)$$

$$\text{(Transitivity): } ((p \equiv q) \wedge (q \equiv r)) \Rightarrow (p \equiv r).$$

Notation Write $x, y, z = m, n, r$ as an abbreviation for $x = m \wedge y = n \wedge z = r$. I also write $p \wedge q \wedge r$ as p, q, r , using a comma (,) to separate the conjuncts when it is convenient.

2.5.3 Additional Aspects of Propositional Logic

2.5.3.1 Omitting Parentheses with Transitive and Associative Operators

It is customary to write $x > y > z$ to mean $(x > y) \wedge (y > z)$, from which $x > z$ can be deduced, because $>$ is a transitive operator. And, it is customary to write $a + b + c$ without parentheses because $+$ is an associative operator. These two conventions conflict when the operator is both transitive and associative, as in the case of the equivalence ($\equiv$) operator. How do we interpret $p \equiv q \equiv r$; is it $(p \equiv q) \wedge (q \equiv r)$, or $(p \equiv q) \equiv r$, which has the same value as $p \equiv (q \equiv r)$? Different interpretations yield different results. For example, with $p, q, r = \text{true}, \text{false}, \text{false}$, the value of $(p \equiv q) \wedge (q \equiv r)$ is *false* whereas that of $(p \equiv q) \equiv r$ is *true*.

There is no “correct” answer. I adopt a common convention in mathematics and logic⁴:

Convention about transitive and associative operators An expression $p_0 \star \dots \star p_n$, where $n > 1$ and $\star$ is a transitive and associative operator, is interpreted as if $\star$ is just *transitive*.

So, the given expression is $(p_0 \star p_1) \wedge (p_1 \star p_2) \dots (p_{n-1} \star p_n)$.

2.5.3.2 Strengthening, Weakening Predicates

Given $p \Rightarrow q$, I say p is *stronger than* q and q is *weaker than* p , though the correct terminology should be p is stronger than or equal to q and q is weaker than or equal to p , yet the incorrect terminology is commonly used. So, *false* is the strongest and *true* the weakest predicate. It is often required to *strengthen* or *weaken* a given predicate, which amounts to finding another predicate that is stronger or weaker than the given one. Predicate $p \wedge q$ is stronger than p and $p \vee q$ weaker than p . We derive from the monotonicity law of the implication operator that:

$$\begin{aligned} (p \Rightarrow q) &\Rightarrow (p \wedge r \Rightarrow q \wedge r), \text{ and} \\ (p \Rightarrow q) &\Rightarrow (p \vee r \Rightarrow q \vee r). \end{aligned}$$

That is, both the left and the right sides of an implication can be strengthened or weakened by the same predicate.

2.5.3.3 Relationship with Set Algebra

The laws of propositional logic correspond to the laws of set algebra as follows. Replace *true*, *false*, $\wedge$, $\vee$, $\neg$, $\equiv$ and $\Rightarrow$ by U (universe), ϕ , $\cap$, $\cup$, complement, $=$ and $\subseteq$, respectively, to transform a law in one system to the other. Thus, the distributivity law of boolean algebra, $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$, yields $p \cap (q \cup r) = (p \cap q) \cup (p \cap r)$.

4. Dijkstra and Scholten [1989] adopt the opposite convention in their book. It yields a number of beautiful identities involving the equivalence operator. A particularly striking example is the “Golden Rule” linking the $\wedge$ and $\vee$ operators: $p \wedge q \equiv p \equiv q \equiv p \vee q$.

2.5.3.4 Boolean Algebra as a Commutative Ring

This section contains advanced material. It is not used elsewhere in the book.

An alternative approach to propositional logic, in [Rocha and Meseguer \[2008\]](#), is to start with two operators: exclusive-or (written $\oplus$) and $\wedge$. Exclusive-or is the negation of equivalence: $p \oplus q$ is *true* iff p and q are different in value. It is also addition (mod 2) by encoding *false* by 0 and *true* by 1. In algebraic terms these two operators form a commutative ring satisfying the following axioms.

- Both $\oplus$ and $\wedge$ are commutative and associative.
- (*false* is zero of $\oplus$) $\text{false} \oplus p = p$.
- (Inverse of $\oplus$) $p \oplus p = \text{false}$.
- (Zero and one of $\wedge$) $\text{false} \wedge p = \text{false}$, $\text{true} \wedge p = p$.
- (Idempotence of $\wedge$) $p \wedge p = p$.
- (Distributivity of $\wedge$ over $\oplus$) $p \wedge (q \oplus r) = (p \wedge q) \oplus (p \wedge r)$.

The other propositional operators are defined in terms of $\oplus$ and $\wedge$:

- $\neg p = p \oplus \text{true}$
- $p \vee q = (p \wedge q) \oplus p \oplus q$
- $p \Rightarrow q = \neg(p \oplus (p \wedge q))$
- $p \equiv q = \neg(p \oplus q)$

2.5.3.5 Functional Completeness of Boolean Operators

A set of boolean operators is *functionally complete* if every boolean function can be expressed using only these operators over boolean operands. I claim that $\{\neg, \wedge, \vee\}$ is a functionally complete set. Given a boolean function whose value is defined by a truth table, I show how to express it using $\{\neg, \wedge, \vee\}$. I explain the procedure using an example, which is easily formalized.

Consider function f over operands p , q and r defined in the last column of Table 2.4. The function value is *true* (written as T) only in the numbered rows. That is, f is *true* iff the operands have values as prescribed in either row (1), (2) or (3). The values of the operands in (1), (2) and (3) are $(\neg p \wedge \neg q \wedge r)$, $(\neg p \wedge q \wedge r)$ and $(p \wedge q \wedge r)$, respectively. So, f is *true* iff any one (or more) of these expressions is *true*. This fact is written as:

$$f \equiv (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (p \wedge q \wedge r).$$

Though the given set $\{\neg, \wedge, \vee\}$ is functionally complete, it is not minimal. For operands x and y , $x \wedge y$ is $\neg(\neg(x \wedge y))$, and using De Morgan's law $x \wedge y$ is $\neg(\neg x \vee \neg y)$. Since all occurrences of $\wedge$ can be removed, $\{\neg, \vee\}$ is functionally complete. Using similar arguments $\{\neg, \wedge\}$ is functionally complete. Also, $\{\neg, \Rightarrow\}$ is functionally

Table 2.4 A truth table for a boolean function over operands p, q and r

p	q	r	f
F	F	F	F
F	F	T	T (1)
F	T	F	F
F	T	T	T (2)
T	F	F	F
T	F	T	F
T	T	F	F
T	T	T	T (3)

complete because $(p \vee q) \equiv (\neg p \Rightarrow q)$. From Section 2.5.3.4 $\{\oplus, \wedge\}$ is functionally complete. In general, given a functionally complete set C if any operator σ in C can be expressed using the set of operators D , then $(C - \{\sigma\}) \cup D$ is a functionally complete set.

Binary operators *nand* (written as $\bar{\wedge}$) and *nor* (written as $\bar{\vee}$) have the following definitions: nand: $p \bar{\wedge} q = \neg(p \wedge q)$, nor: $p \bar{\vee} q = \neg(p \vee q)$.

Each of these operators is functionally complete:

(1) for nand, $\neg p = p \bar{\wedge} p$, so $p \wedge q = \neg(p \bar{\wedge} q)$, and

(2) for nor, $\neg p = p \bar{\vee} p$, so $p \vee q = \neg(p \bar{\vee} q)$.

De Morgan's law can be applied to define $\wedge$ and $\vee$ directly using $\bar{\wedge}$ and $\bar{\vee}$ (see Figure 2.9).

$$\begin{array}{lcl}
 \neg p & = & p \bar{\wedge} p \\
 p \wedge q & = & (p \bar{\wedge} q) \bar{\wedge} (p \bar{\wedge} q) \\
 p \vee q & = & (p \bar{\wedge} p) \bar{\wedge} (q \bar{\wedge} q)
 \end{array}
 \quad \parallel \quad
 \begin{array}{lcl}
 \neg p & = & p \bar{\vee} p \\
 p \wedge q & = & (p \bar{\vee} p) \bar{\vee} (q \bar{\vee} q) \\
 p \vee q & = & (p \bar{\vee} q) \bar{\vee} (p \bar{\vee} q)
 \end{array}$$

Figure 2.9 Completeness of nand and nor operators.

Boolean operators are called *gates* in computer circuit design. In the early days of computing, it was easier to manufacture circuits using just one type of gate. The guidance computer for the Apollo mission used only nor gates. Modern computers use multiple kinds of gates, corresponding to all the operators I have introduced.

2.5.4 Satisfiability, Validity

A boolean proposition is *satisfiable* if it is possible to assign values to its variables so that the proposition value is *true*; the proposition is *unsatisfiable* otherwise. A proposition is *valid*, or a *tautology*, if for all possible assignment of values to its variables the proposition value is *true*; otherwise, it is *invalid*. Therefore, proposition P is valid iff $\neg P$ is unsatisfiable. To show that a conclusion c follows from a set

of premises $p_0, p_1, \dots, p_n$, we need to show that $(p_0 \wedge p_1 \wedge \dots \wedge p_n) \Rightarrow c$ is valid, or that its complement $(p_0 \wedge p_1 \wedge \dots \wedge p_n) \wedge \neg c$ is unsatisfiable.

A fundamental problem of computer science is proving satisfiability (or, equivalently, validity) of boolean propositions, commonly known as the SAT-solving problem. The more general analogous problem in predicate calculus is called theorem proving. There is a simple algorithm for solving propositional satisfiability: just check every possible combination of variable values to determine if any satisfies the proposition. For a proposition with n variables there are 2^n possible combination of values. So, this exhaustive search algorithm is effective only for small values of n . Unfortunately, we know nothing better for the worst case. Yet, there are now several algorithms that are very effective in practice. I discuss two such algorithms, Resolution and DPLL, in this section.

Many problems in combinatorial mathematics, and in hardware and software design, are now solved using SAT solvers. Heule et al. [2016] solved the “Boolean Pythagorean triples” problem using a SAT-solver and validated a proof of the solution that was 200 terabytes in size.

2.5.4.1 Conjunctive and Disjunctive Normal Form

A conjunctive clause, or *conjunct*, is of the form $p_0 \wedge p_1 \wedge \dots \wedge p_i \dots \wedge p_n$ where each p_i is a literal, that is, a variable or negation of a variable. A disjunctive clause, or *disjunct*, is of the form $p_0 \vee p_1 \vee \dots \vee p_i \dots \vee p_n$ where each p_i is a literal. An empty conjunctive clause has value *true* and empty disjunctive clause is *false*.

A boolean proposition or function that is a conjunction of a set of disjuncts is in *conjunctive normal form* (CNF), and a proposition that is a disjunction of a set of conjuncts is in *disjunctive normal form* (DNF). For example, the following proposition is in CNF. The notation “ $P::$ ” used below means that P can be used to refer to the given proposition.

$$P:: (\neg x \vee y \vee z) \wedge (x \vee \neg z) \wedge (\neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee y \vee z).$$

The negation of this proposition, applying De Morgan’s law, is in DNF:

$$\neg P:: (x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge z) \vee (y \wedge \neg z) \vee (x \wedge y \wedge z) \vee (\neg x \wedge \neg y \wedge \neg z).$$

It can be shown that any boolean proposition can be written equivalently in a unique way in CNF, and also in DNF, up to the rearrangements of the conjuncts and disjuncts. A complete proof appeals to induction, see Chapter 3. Here I give a sketch of a recursive algorithm to convert a proposition to equivalent CNF and DNF. I examine the various forms of a boolean proposition and the steps needed to construct its CNF and DNF in each case.

1. The proposition is a literal: then the proposition is already, vacuously, in CNF and DNF.
2. The proposition is $P \vee Q$ where each of P and Q is a proposition:

To construct the DNF, construct the DNF of P and Q in P' and Q' , respectively. Then $P' \vee Q'$ is in DNF.

To construct the CNF, construct the CNF of P and Q in P'' and Q'' , respectively. Let $P'' = p_0 \wedge p_1 \wedge \cdots \wedge p_i \cdots \wedge p_m$ and $Q'' = q_0 \wedge q_1 \wedge \cdots \wedge q_i \cdots \wedge q_n$, where each p_i and q_j is a disjunct. Then

$$P \vee Q = (p_0 \wedge p_1 \wedge \cdots \wedge p_i \cdots \wedge p_m) \vee (q_0 \wedge q_1 \wedge \cdots \wedge q_i \cdots \wedge q_n).$$

Applying distributivity of $\vee$ over $\wedge$

$$P \vee Q = \bigwedge_{0 \leq i \leq m, 0 \leq j \leq n} (p_i \vee q_j)$$

Each $(p_i \vee q_j)$ is a disjunct because p_i and q_j are both disjuncts. Therefore, this proposition is in CNF.

3. The proposition is $P \wedge Q$ where each of P and Q is a proposition: The steps are dual of those in converting $P \vee Q$.

For CNF, construct CNF, P' and Q' , respectively, of P and Q . Then $P' \wedge Q'$ is in CNF.

For DNF, construct DNF, P'' and Q'' , respectively, of P and Q . Next, apply distributivity of $\wedge$ over $\vee$ to construct the DNF.

4. The proposition is $\neg P$ where P is a proposition:

To construct the DNF, first construct the CNF P' of P . Apply De Morgan's law to $\neg P'$ to obtain the equivalent DNF.

To construct the CNF, first construct the DNF P'' of P . Apply De Morgan's law to $\neg P''$ to obtain the equivalent CNF.

Reduction of CNF proposition by assigning specific truth values I show the reduction of a proposition in CNF when one of its literals is assigned a truth value. Such reductions are used in describing the resolution principle and the DPLL algorithm later in this section.

Let P be a proposition given in CNF and $P[x := v]$ be the proposition derived from P by setting variable x to the truth value v , *true* or *false*. Setting any literal x to *true* has the following effect: (1) eliminate a clause in which literal x appears because the clause becomes *true*, and (2) eliminate $\neg x$ from every clause because its value is *false*. Observe that $P[x := v]$ is in CNF, it includes neither x nor $\neg x$, and it has no more clauses than P .

As an example, consider the proposition

$$P:: (\neg x \vee y \vee z) \wedge (x \vee \neg z) \wedge (\neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee y \vee z).$$

Then

$$P[x := \text{true}]:: (y \vee z) \wedge (\neg y \vee z) \wedge (\neg y \vee \neg z), \text{ and}$$

$$P[x := \text{false}]:: (\neg z) \wedge (\neg y \vee z) \wedge (y \vee z).$$

2.5.4.2 Resolution Principle

Resolution principle, originally due to [Davis and Putnam \[1960\]](#) and later modified by [Robinson \[1965\]](#), is a powerful theorem proving method. Here I discuss its simple variant applied to boolean propositions.

A given proposition P in CNF can be regarded as a set of disjunctive clauses where the conjunction of the clauses is implicit. An empty clause denotes *false*. Consider two clauses of the form $p \vee q$ and $\neg q \vee r$ in P where q and $\neg q$ are complementary literals and p and r are disjunctions of possibly several literals. We can deduce that $p \vee r$, called a *resolvent*, follows from these two clauses:

$$\begin{aligned} & (p \vee q) \wedge (\neg q \vee r) \\ \Rightarrow & \{\text{distribute } \wedge \text{ over } \vee\} \\ & (p \wedge \neg q) \vee (p \wedge r) \vee (q \wedge \neg q) \vee (q \wedge r) \\ \Rightarrow & \{\text{eliminate } (q \wedge \neg q)\} \\ & (p \wedge \neg q) \vee (p \wedge r) \vee (q \wedge r) \\ \Rightarrow & \{\text{each of the first two clauses imply } p, \text{ the last clause } r\} \\ & p \vee r \end{aligned}$$

This inference rule is the *resolution rule*. The resolvent $p \vee r$ is then added to the set of clauses in P , and the procedure is repeated until the resolution rule can no longer be applied. The resolvent could be the empty clause, derived from clauses p and $\neg p$.

The resolution principle claims that this procedure eventually either creates (1) an empty resolvent or (2) a *closed* set of clauses P^* so that the resolution rule does not create any new resolvent. In the first case P is unsatisfiable, and in the second case P is satisfiable. Observe that repeated application of resolution terminates, simply because there are a finite number of disjunctive clauses over a finite number of literals.

If the procedure claims that P is unsatisfiable, then P is indeed unsatisfiable, a property called *soundness*. Conversely, if P is unsatisfiable, the procedure will eventually claim it to be unsatisfiable, a property called *completeness*. I prove these two claims below.

The resolution rule is simple to apply in a theorem prover because there is just one rule to apply in every step instead of choosing a rule from a number of possible ones. It has indeed proved very effective in propositional theorem proving.

Two additional simple rules are applied during the procedure: (1) any clause that contains a variable and its negation, $p \vee \neg p$, is a tautology, so remove it from P , and (2) eliminate a *unit clause*, a clause that contains a single literal, by assigning *true* to that literal. The first rule reduces the size of P . The second rule converts P to $P[x := v]$ by assigning v , *true* or *false*, to some variable x .

Example 2.5 I show that the following CNF is unsatisfiable:

$$(p \vee \neg q) \wedge (\neg p \vee \neg q) \wedge (p \vee \neg r) \wedge (\neg p \vee \neg r) \wedge (q \vee r) \wedge (\neg q \vee \neg r).$$

$\neg q$	from $(p \vee \neg q) \wedge (\neg p \vee \neg q)$
r	from $q \vee r$ and above
$\neg r$	from $(p \vee \neg r) \wedge (\neg p \vee \neg r)$
<i>false</i>	from above two

Soundness of the resolution procedure I show that if the procedure ever generates an empty clause from P , then P is unsatisfiable, that is, its value is *false*. An application of the resolution rule to $p \vee q$ and $\neg q \vee r$ in P creates $p \vee r$; let $P' = P \cup \{p \vee r\}$. We note that $P \equiv P'$ because: (1) $P \Rightarrow P'$, which follows from $(p \vee q) \wedge (\neg q \vee r)$ implies $p \vee r$, and (2) $P' \Rightarrow P$ because P' includes an extra clause, so it is stronger than P . Therefore, P^* at termination is equivalent to P .

Since P^* includes the empty clause, whose value is *false*, P^* is *false*; so is P .

Completeness of the resolution procedure I prove the contrapositive of the completeness claim: if the procedure terminates with a closed P^* that does not include the empty clause, it is satisfiable.⁵ I show a satisfying assignment of variable values in P .

Observe that P^* does not include both x and $\neg x$ as clauses for any x , for then resolution can be applied to these clauses, and P^* is closed. Further, that all tautologies and unit clauses have been eliminated. Choose a variable x arbitrarily and assign an arbitrary truth value v to x . Note that $P^*[x := v]$ does not include the empty clause because that can happen only if x with value $\neg v$, or $\neg x$ with value v , is a clause in P^* , and we have assumed that there are no unit clauses in P^* . Next, eliminate any tautologies and unit clauses.

I claim that $P^*[x := v]$ is closed (see Exercise 14, page 76), so no resolution step can be applied to it. Further, $P^*[x := v]$ has fewer variables because it does not have

5. I owe this proof to Vladimir Lifschitz.

x . Repeat the assignment procedure with $P^*[x := v]$ until the proposition has no variable, hence no clause. The assigned truth values satisfy P^* , so P^* is satisfiable. Since $P \equiv P^*$, P is satisfiable.

2.5.4.3 DPLL Algorithm

The Davis–Putnam–Logemann–Loveland algorithm, abbreviated as DPLL, is described in Davis et al. [1962]. It and its variants are used extensively for SAT-solving. It uses a few simple observations about boolean algebra, yet it is surprisingly powerful in practice.

DPLL is a backtracking algorithm. Given proposition P , assign value *true* to an arbitrary variable x in it. Proposition $P[x := \text{true}]$ has fewer variables than P because x does not occur in it. Check for its satisfiability recursively; if it is satisfiable, then so is P with x having value *true*. If it is unsatisfiable, check $P[x := \text{false}]$ for satisfiability. If it is unsatisfiable, then P is unsatisfiable, otherwise P is satisfiable with x having value *false*. This procedure does not require P to be in a normal form, such as CNF or DNF; however, the refinement of the algorithm applies to propositions in CNF.

Figure 2.10 shows application of the procedure in successive stages on a given proposition. Assigning a value to a variable causes elimination of clauses and literals from clauses. So, eventually, either (1) the proposition becomes empty, hence satisfiable, or (2) there is an empty clause, hence the proposition is unsatisfiable. A proposition is satisfiable iff there is a leaf with an empty proposition, shown by T in the figure. The assignment of values to variables along the path to this leaf is a satisfying assignment. In this example p , $\neg q$ and r are true in a satisfying assignment.

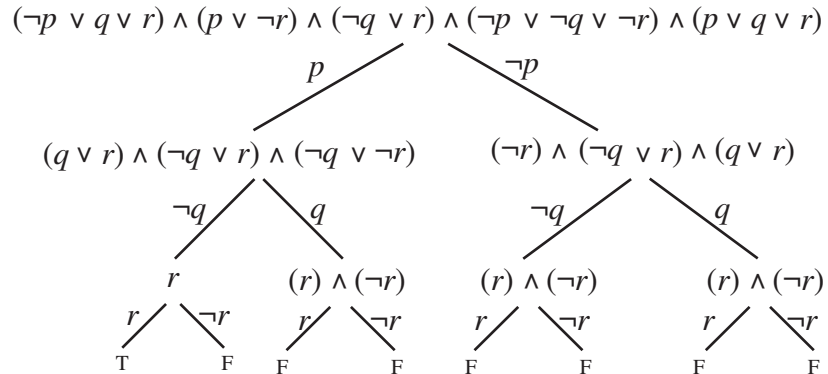


Figure 2.10 Successive reduction of a proposition.

The given version of the algorithm is very inefficient if the proposition is unsatisfiable, then all leaf nodes have to be explored at a cost of 2^n for n variables. What makes DPLL effective are the following heuristics.

- Unit propagation: If P has a clause that has just one literal, say x , then x has to be *true* in any satisfying assignment. So, there is no need to explore the proof of $P[x := \text{false}]$. In many cases, unit propagation causes further unit propagation. In Figure 2.10 proposition $(\neg r) \wedge (\neg q \vee r) \wedge (q \vee r)$ has a unit literal $(\neg r)$; so, r has to be set *false*.
- Unipolar literal: Literal x is *unipolar* if $\neg x$ does not appear in any clause. Then assign *false* to x so that every clause in which it occurs becomes *true*; so, these clauses can be eliminated.

DPLL is extremely effective in practice by judicious applications of these heuristics, choice of the next variable to be assigned a value, and by using appropriate data structures.

2.6 Predicate Calculus

A propositional predicate cannot express a statement like “for every number there is a larger number”. Predicate calculus is a generalization of propositional logic. It includes universal quantification, denoted by $\forall$ which is a generalization of $\wedge$, and existential quantification, denoted by $\exists$ which is a generalization of $\vee$, over variables. The given statement is typically written $\forall x. \exists y. y > x$, and read as “for every x there exists a y such that $y > x$ ”.

2.6.1 Free and Bound Variables

In the predicate $(\exists x. 0 \leq x < 10. z > x)$, variable x is *bound* and z is *free*. The meaning of a bound variable is specified in the predicate; the meaning of a free variable depends on the context. For example, z may be a variable in a program and the given predicate an assertion at a specific program point. Bound variables may be systematically renamed using a fresh variable name (or a literal for a boolean variable), so $\forall x. \exists y. y > x$ is the same as $\forall t. \exists x. x > t$, renaming x by t and then y by x . You must be careful in renaming. Never rename a bound variable by the name of an existing variable in the proposition. In this example, you should not rename y by x and then x by t .

2.6.2 Syntax of Quantified Expressions

I extend the traditional syntax of quantified expressions in order to make the ranges of the bound variables explicit and adopt this notation for writing quantified expressions over other types of operands including arithmetic expressions

(see Section 2.6.5, page 53). For example, $(\exists x. 0 \leq x < 10. z > x)$ is written as $(\exists x : 0 \leq x < 10 : z > x)$, which consists of three parts separated by colons ($:$) as follows: the quantifier and the bound variable x come first, the range of the bound variable is given next by the predicate $0 \leq x < 10$, and the main predicate $z > x$, over the free variable z and the bound variable x , comes last.

Write a quantified expression in the following form: $(\otimes x : q(x) : e(x))$. Here, $\otimes$ is either $\forall$ or $\exists$, x a list *bound* variables, $q(x)$ a predicate that determines the *range* of the bound variables, and $e(x)$ a predicate called the *body*. If the range is implicit, then write $(\otimes x :: e(x))$. The value of the quantified expression is the result of applying $\otimes$ to the set $\{e(x) \mid q(x)\}$. In words, compute the set of all $e(x)$ where x satisfies $q(x)$; then apply $\otimes$ to this set. If the range given by $q(x)$ is empty, the value of the expression is the *unit element* of operator $\otimes$. Unit element of $\forall$ (i.e., $\wedge$) is *true* and of $\exists$ (i.e., $\vee$) is *false*.

Example 2.6 I explain this notation with some examples. Formalize the statement “for every number there is a larger number”. I write a series of equivalent expressions, where each but the final expression is a mixture of informal and formal statements. Below each variable is quantified over numbers.

$(\forall x :: \text{there is a number } y \text{ larger than } x)$, or
 $(\forall x :: (\exists y :: y \text{ is larger than } x))$, or
 $(\forall x :: (\exists y :: y > x))$.

To show the ranges of the variables explicitly, write:

$(\forall x : x \in \text{number} : (\exists y : y \in \text{number} : y > x))$.

The statement “there is a unique element satisfying predicate q ” can be expressed by:

$(\exists x :: q(x)) \wedge (\forall x, y : q(x) \wedge q(y) : x = y)$.

Expression $(\forall i : 0..N : A[i] \leq A[i + 1])$ is a statement about the elements of array A indexed from 0 to N , that each element is smaller than or equal to the next element in the array. That is, the elements of A are sorted in ascending order, or A is empty ($N \leq 0$). Analogously, $(\exists i : 0..N : A[i] \leq A[i + 1])$ is *true* iff there is an array element within the interval $0..N$ whose value is no more than its next element; the statement is *false* if there is no such element, in particular if the range is empty ($N \leq 0$).

2.6.3 Laws of Predicate Calculus

- (Empty Range)
 $(\forall i : \text{false} : b) \equiv \text{true}$
 $(\exists i : \text{false} : b) \equiv \text{false}$

- (Trading)

$$(\forall i : q : b) \equiv (\forall i :: q \Rightarrow b)$$

$$(\exists i : q : b) \equiv (\exists i :: q \wedge b)$$
- (Moving) Given that i does not occur as a free variable in p ,

$$p \vee (\forall i : q : b) \equiv (\forall i : q : p \vee b)$$

$$p \wedge (\exists i : q : b) \equiv (\exists i : q : p \wedge b)$$
- (De Morgan's laws)

$$\neg(\exists i : q : b) \equiv (\forall i : q : \neg b)$$

$$\neg(\forall i : q : b) \equiv (\exists i : q : \neg b)$$
- (Range weakening) Given that $q \Rightarrow q'$,

$$(\forall i : q' : b) \Rightarrow (\forall i : q : b)$$

$$(\exists i : q : b) \Rightarrow (\exists i : q' : b)$$
- (Body weakening) Given that $b \Rightarrow b'$,

$$(\forall i : q : b) \Rightarrow (\forall i : q : b')$$

$$(\exists i : q : b) \Rightarrow (\exists i : q : b')$$

A number of identities can be derived from the trading rule [see [Gries and Schneider 1994](#), chapter 9]; I show two below.

$$(\forall i : q \wedge r : b) \equiv (\forall i : q : r \Rightarrow b)$$

$$(\exists i : q \wedge r : b) \equiv (\exists i : q : r \wedge b)$$

The following versions of the Moving rule hold iff range q is not *false*.

$$p \wedge (\forall i : q : b) \equiv (\forall i : q : p \wedge b)$$

$$p \vee (\exists i : q : b) \equiv (\exists i : q : p \vee b)$$

2.6.4 Duality Principle

Given boolean-valued functions f and g over a list of variables X , f is a dual of g if $f(\neg X) = \neg g(X)$; here $\neg X$ denotes the negated list of variables of X . It can be shown that if f is a dual of g , then g is a dual of f , and the dual of any function is unique (see Exercises 17(a) and (b), page 78). Therefore, the dual of the dual of a function is the function itself.

The following results about duality are easily established from the definition. The two 0-ary, or constant, boolean-valued functions, *true* and *false*, are duals of each other. There are four 1-ary boolean-valued functions. These include *true* and *false* over one argument, which are again duals. The other two functions are negation ($\neg$) and the identity function which are self-duals, that is, each is its own dual.

Binary boolean-valued functions are typically written as infix operators, as in $x \oplus y$. Dual $\oplus$ of a binary operator $\otimes$ is uniquely defined by: $(x \oplus y) \equiv \neg(\neg x \otimes \neg y)$, from the definition of dual. Exercise (17c) (page 78) asks you to show that the following pairs of operators are duals: $(\wedge, \vee)$, $(=, \neq)$, $(\equiv, \not\equiv)$. Even though quantifiers are not functions, universal $(\forall)$ and existential $(\exists)$ quantifiers are taken to be duals.

Dual of a boolean expression p , written as p' , is obtained by replacing each function in p by its dual. The dual of $(\neg x \vee y) \wedge (x \vee \neg z)$, for instance, is $(\neg x \wedge y) \vee (x \wedge \neg z)$.

The *duality principle* says that $(p = q) \equiv (p' = q')$, where p and p' , and q and q' are duals. That is, $p = q$ is valid iff $p' = q'$ is valid. This principle is quite useful. For example, the distributivity laws

$$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r),$$

$$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r),$$

are duals of each other. So are the pair of De Morgan's laws.

To justify this principle, prove this proposition where the list of variables, X , is shown explicitly: $(p(X) = q(X)) \equiv (p'(X) = q'(X))$.

$$\begin{aligned} & p(X) = q(X) \\ \equiv & \text{ \{negate both operands\} } \\ & \neg p(X) = \neg q(X) \\ \equiv & \text{ \{ } \neg p(X) = p'(\neg X), \neg q(X) = q'(\neg X) \text{ \} } \\ & p'(\neg X) = q'(\neg X) \\ \equiv & \text{ \{replace variables } X \text{ by their negations\} } \\ & p'(X) = q'(X) \end{aligned}$$

2.6.5 Quantified Expressions over Non-Boolean Domains

I extend the notation for quantification, defined for logical expressions in Section 2.6.2 (page 50), to other domains such as arithmetic expressions. A quantified expression is of the form $(\otimes x : q(x) : e(x))$ where $\otimes$ is a *commutative and associative* binary operator that can be applied to pairs of values in the given domain. The value of expression $(\otimes x : q(x) : e(x))$ is computed by applying $\otimes$ pairwise to the values in the set $\{e(x) \mid q(x)\}$ in arbitrary order.

In the domain of numbers $\otimes$ is typically $+$, $\times$, $\min$ or $\max$, for boolean domain $\wedge$, $\vee$ or $\equiv$, and for the domain of sets $\cup$ or $\cap$. Operator *id* is the identity function applied to a set, so $(id\ x : q(x) : e(x)) = \{e(x) \mid q(x)\}$.

If the range given by $q(x)$ is empty, $\{e(x) \mid q(x)\}$ is empty. The value of the expression then is the unit element of operator $\otimes$. Unit elements of the typical operators are shown below within parentheses. Below U denotes the universal set of discourse.

$$+ (0), \times (1), \min (+\infty), \max (-\infty), \wedge (true), \vee (false), \equiv (true), id (\{\}), \cup (\{\}), \cap (U).$$

In most, but not all, arithmetic expressions that use $\sum$ or $\prod$, I use quantified expressions with operators $+$ or $\times$. For example, $\sum_{0 \leq i}^n (1/2^i)$ can now be written as $(+i : 0 \leq i \leq n : 1/2^i)$.

Example 2.7 Consider the following examples of quantified expressions over arrays and matrices. Here, i and j are integers from 0 to $N - 1$ where N is specified elsewhere, B a matrix of booleans, A an array of numbers and M a diagonal matrix of numbers.

1. $(\forall i, j : 0 \leq i < N \wedge 0 \leq j < N \wedge i \neq j : B[i, j])$.

The expression value is *true* iff all off-diagonal elements of matrix $B[0..N, 0..N]$ are *true*.

Recall the convention from Section 2.1.2.5 (page 9) that $s..t$ is the sequence of integers that includes s but excludes t . So, we can rewrite this expression as $(\forall i, j : i \in 0..N, j \in 0..N, i \neq j : B[i, j])$.

2. $(+i : i \in 0..N : A[i])$.

The expression value is $\sum_{i=0}^{N-1} A[i]$.

3. $(\min i : i \in 0..N \wedge (\forall j : j \in 0..N : M[i, j] = 0) : i)$.

The expression value is the index of the row in M all of whose elements are zero, and whose index is the smallest among all such rows; if there is no such row, the expression value is ∞ . ■

I extend the notation to apply to sequences instead of sets; then $\otimes$ need only be a binary associative operator. The value of $(\otimes x : q(x) : e(x))$ is computed by applying $\otimes$ pairwise to the values in $\langle e(x) \mid q(x) \rangle$ in order. For example, given that $str[0..N]$ is an array of strings, $(++ i : i \in 0..N : str[i])$ is the concatenation of all the strings in the array. For an empty sequence, for instance $N = 0$ in this example, the expression value is the unit of $\otimes$; for the concatenation operator the unit element is the empty sequence. Another example of $\otimes$ is matrix multiplication whose unit element is the identity matrix.

2.7 Formal Proofs

Since time immemorial people have used common sense reasoning in their every-day affairs. Statements such as “a stone falls to earth because earth is the natural home of the stone” were traditionally based on common sense of that time. Rhetorical reasoning, used for political arguments, often (deliberately) confuse $p \Rightarrow q$ with $q \Rightarrow p$, or ascribe causation when there is only correlation between events.

Common sense, that is, informal reasoning was employed in the physical sciences before the advent of calculus. Computer programs are viewed by many as physical processes evolving over time during their executions, and arguments

about their behavior are often informal. Programmers mostly use informal reasoning for program design, coding and experimental validation of programs. Such reasoning has many virtues including its reliance on intuition and avoidance of formal mathematics so that it appeals to a larger readership. However, it is error-prone as it typically ignores the corner cases that arise far too often in actual practice with computers. A common fallacy in the early days of multiprocessor programming was to argue that a processor will execute at least one step of its program if another processor has already taken a million steps, say. This makes perfect sense if we regard each processor as a human being.

Paradoxically, common sense reasoning may be *more* difficult to understand and laborious to construct than a proof that relies on formal tools. Consider this small problem: find all numbers whose square is equal to the number itself. Informally, we proceed by noting that since the square is non-negative the number is itself is non-negative. The first non-negative number is 0 and, clearly, 0 is a solution. Between 0 and 1, the square is smaller than the number itself (multiplying by x , where $0 < x < 1$, reduces any positive number). Thus, there is no x , $0 < x < 1$, that is a solution. Beyond 1, any multiplication increases a number, so no such number is a solution. So, the only remaining case to consider is 1, and it is indeed a solution. The formal approach, based on algebra, solves the equation $x^2 = x$, that is, $x^2 - x = 0$, or $x(x - 1) = 0$. This has the solutions $x = 0$ and $x - 1 = 0$, or $x = 1$.

Formal logic plays the role of algebra in many of our proofs. We simplify our thinking by relying on the rules of logic to make deductions. Logic makes the arguments not only rigorous but often simpler. I do use common sense reasoning for many problems but supplant it with formal reasoning for the more complicated cases. Logic is a very effective tool to justify intuition.

2.7.1 Proof Strategies

In a number of ways, constructing a proof is similar to constructing a program; in fact, there is a formal correspondence between programs and proofs. There is no recipe for program construction, just guidelines; it is similar with proofs. I discuss a few heuristics for proof construction in this section.

A formal proof starts with a number of facts that are assumed to be true, often called *axioms*, *antecedents*, *hypotheses* or *premises*. The proof ends with a *conclusion*, *result* or *consequence*. Taking p to stand for the conjunction of all the premises and q for the conjunction of all the conclusions, a proof is a demonstration of $p \Rightarrow q$ using the laws of logic.

Every law of logic gives a strategy for constructing a proof. For example, the law $(p \equiv q) = (p \Rightarrow q) \wedge (q \Rightarrow p)$ gives a strategy for proving an equivalence by mutual implication, and the strategy given by the rule $p \equiv (\neg p \Rightarrow \text{false})$ is commonly

known as a *proof by contradiction*: to prove p assume the contrary, that p is *false*, and then derive a contradiction. I identify a few common strategies below and show examples of their use in Section 2.8.

1. Elementary proof: Prove a result in a single step from a hypothesis.
2. Transitivity: Given $p \Rightarrow q$ and $q \Rightarrow r$, conclude $p \Rightarrow r$. In most cases, we will have a chain of implications, $p \Rightarrow q_0, q_0 \Rightarrow q_1, \dots, q_n \Rightarrow q$, from which we deduce $p \Rightarrow q$.
3. Conjunction and disjunction: This strategy allows proving two simpler results that are then combined to arrive at the conclusion. The conjunction rule is based on: From $p \Rightarrow q$ and $p \Rightarrow q'$, deduce $p \Rightarrow q \wedge q'$. So, in order to prove $p \Rightarrow q \wedge q'$, prove $p \Rightarrow q$ and $p \Rightarrow q'$ separately.
The disjunction rule is based on: From $p \Rightarrow q$ and $p' \Rightarrow q$, deduce $p \vee p' \Rightarrow q$. So, $p \vee p' \Rightarrow q$ can be proved by proving $p \Rightarrow q$ and $p' \Rightarrow q$.
A special case of the disjunction rule is known as “proof by case analysis”: From $p \Rightarrow q$ and $\neg p \Rightarrow q$, deduce q .
4. Mutual implication: Proof by mutual implication establishes $p \equiv q$ by showing $p \Rightarrow q$ and $q \Rightarrow p$.
5. Contrapositive: The contrapositive rule $(p \Rightarrow q) = (\neg q \Rightarrow \neg p)$ allows proving $p \Rightarrow q$ by proving $\neg q \Rightarrow \neg p$.
6. Contradiction: In order to prove p , prove $\neg p \Rightarrow \text{false}$. From the contrapositive rule, $(\neg p \Rightarrow \text{false}) = (\text{true} \Rightarrow p)$, so p is established.
7. Universally quantified proof, prove $(\forall x : x \in S : p(x))$ for a given set S : If S is finite and small, prove the result by enumerating each x in S and proving $p(x)$. For infinite or large S , either employ induction (see Chapter 3) or use contradiction: for any arbitrary member a of S , show that $\neg p(a)$ implies *false*.
8. Existentially quantified proof, prove $(\exists x : x \in S : p(x))$ for a given set S : A *constructive* strategy is to display a “witness” w , a member of the set, for which $p(w)$ holds. For example, to show that there is a prime number below 5, that is, $(\exists x : 1 < x < 5 : x \text{ is prime})$, display the witness 2 (or 3). A *non-constructive* strategy may not display a specific witness. Instead, postulate a predicate q for which we can show $q \Rightarrow (\exists x : x \in S : p(x))$ and $\neg q \Rightarrow (\exists x : x \in S : p(x))$, and then apply the disjunction rule (see Section 2.8.2, page 61, for an example).

2.7.2 A Style of Writing Proofs

There are many styles of writing proofs ranging from “trivially ...” to one that can be checked automatically. Most proofs fall somewhere in between, typically a mixture

of text interspersed with some mathematics. The proofs in this book are meant to be as simple as possible while being convincing to a reader with the appropriate mathematical and computational background. Almost all proofs in this book are precise. A precise proof is not necessarily formal but can be made formal by following some routine heuristics. There is no need to translate every statement to formal logic; a diagram or table can sometimes be simpler and equally convincing. For entangled arguments, however, it is simpler to rely on a formal or semi-formal proof rather than common sense arguments.

In many cases, I adopt a style for rendering proofs, suggested by W.H.J. Feijen, that makes the steps and justifications more transparent. A proof is written as a sequence of steps, as shown below for a proof of $p \Rightarrow s$. The proof proceeds by assuming p in the top line, then proving q , r and s in sequence. Each proof step includes a justification within braces between its antecedent and conclusion. Using the transitivity of implication, we conclude $p \Rightarrow s$.

$$\begin{array}{l}
 p \\
 \Rightarrow \quad \{ \text{why } p \Rightarrow q \} \\
 q \\
 \Rightarrow \quad \{ \text{why } q \Rightarrow r \} \\
 r \\
 \Rightarrow \quad \{ \text{why } r \Rightarrow s \} \\
 s
 \end{array}$$

Here, $\Rightarrow$ is the main connective in the proof steps. Any transitive relation can be used in place of $\Rightarrow$ as the main connective. The proof of an order relation, $x < y$ for instance, may use $<$ as the connective, or perhaps any or all of $<$, $\leq$ and $=$ as long as there is at least one occurrence of $<$. A proof of equality may use $=$ and equivalence proof may use $\equiv$ as connectives.

The steps and justifications may be excruciatingly small or uncomfortably large, based on the readership. I adopt a granularity for proofs in this book that is comfortable for me, and, hopefully, for my readers. Given below is a proof in great detail.

Example 2.8 Given equivalence relations R_1 and R_2 on set S , show that their intersection R is an equivalence relation. Relation R can be written:

$$x R y \equiv (x R_1 y) \wedge (x R_2 y), \text{ for all } x \text{ and } y \text{ in } S.$$

R is reflexive:

$$\begin{array}{l}
 x R x \\
 \equiv \{ \text{definition of } R \}
 \end{array}$$

$$\begin{aligned}
& x R_1 x \wedge x R_2 x \\
\equiv & \{R_1, \text{being an equivalence relation, is reflexive. Similarly, } R_2\} \\
& \text{true} \wedge \text{true} \\
\equiv & \{\text{logic}\} \\
& \text{true}
\end{aligned}$$

R is symmetric:

$$\begin{aligned}
& x R y \\
\equiv & \{\text{definition of } R\} \\
& x R_1 y \wedge x R_2 y \\
\equiv & \{R_1, \text{being an equivalence relation, is symmetric. Similarly, } R_2\} \\
& y R_1 x \wedge y R_2 x \\
\equiv & \{\text{definition of } R\} \\
& y R x
\end{aligned}$$

R is transitive:

$$\begin{aligned}
& x R y \wedge y R z \\
\equiv & \{\text{definition of } R\} \\
& (x R_1 y \wedge x R_2 y) \wedge (y R_1 z \wedge y R_2 z) \\
\equiv & \{\text{rearranging the conjuncts}\} \\
& (x R_1 y \wedge y R_1 z) \wedge (x R_2 y \wedge y R_2 z) \\
\Rightarrow & \{R_1, \text{being an equivalence relation, is transitive. Similarly, } R_2\} \\
& x R_1 z \wedge x R_2 z \\
\equiv & \{\text{definition of } R\} \\
& x R z
\end{aligned}$$

The reader should note that each proof step is independently justified and the relationships among the facts are captured by the overall proof structure. Such a proof is easy to verify, and it is clear where each hypothesis is used in the proof. Typically, I will not write proofs in such detail, for example just write “predicate calculus” to justify a step that depends on the application of one of the laws given in Sections 2.5.2 and 2.6.3.

Note: In the last proof for the transitivity of R , I used two connectives, $\equiv$ and $\Rightarrow$, to deduce that $(x R y \wedge y R z) \Rightarrow x R z$. I could have used $\Rightarrow$ as the sole connective. Use of two connectives is safe as long as one of the connectives is stronger than the other; in this case $\equiv$ is stronger than $\Rightarrow$, that is, $(p \equiv q) \Rightarrow (p \Rightarrow q)$. I used $\equiv$ to emphasize that a stronger fact can be deduced in an intermediate step. ■

I adopt a slightly different format if the justifications are short. The propositions appear in the left part of each line along with the logical connective, and the justifications in the right side of each line. The logical connective is omitted if a proposition is derived from the propositions that are listed in the justification (see Example 2.5, page 48).

The proof that R is transitive is redone in this style below.

$x R y \wedge y R z$	use definition of R
$\equiv (x R_1 y \wedge x R_2 y) \wedge (y R_1 z \wedge y R_2 z)$	rearrange the conjuncts
$\equiv (x R_1 y \wedge y R_1 z) \wedge (x R_2 y \wedge y R_2 z)$	R_1 and R_2 are transitive
$\equiv x R_1 z \wedge x R_2 z$	definition of R
$\equiv x R z$	

■

In the next section, I give a few small examples that highlight the usefulness of these proof styles. Also, see solutions of Exercises (15) through (17).

2.8 Examples of Proof Construction

The proofs given in this section show the style I will employ in the rest of the book. Some of the proofs are quite formal, where it is simpler to be formal. Other proofs are precise but less formal, where formalism does not add to the simplicity, and the proof can be formalized if needed.

2.8.1 Proofs by Contradiction, Contrapositive

2.8.1.1 $\sqrt{2}$ is Irrational

The proof that $\sqrt{2}$ is irrational is by contradiction: assume $\sqrt{2}$ is rational, and then derive *false*. Let $\sqrt{2}$ be m/n where m and n are integers that have no common factor.

$(\sqrt{2} = m/n), (m, n \text{ have no common factor})$
$\Rightarrow \{\text{squaring}\}$
$(m^2/n^2 = 2), (m, n \text{ have no common factor})$
$\Rightarrow \{\text{arithmetic}\}$
$(m^2 = 2 \times n^2), (m, n \text{ have no common factor})$
$\Rightarrow \{\text{since } m^2 = 2 \times n^2, m \text{ is even, say } m = 2 \times s\}$
$(m = 2 \times s), (m^2 = 2 \times n^2), (m, n \text{ have no common factor})$
$\Rightarrow \{m^2 = 2 \times n^2 \Rightarrow n^2 = m^2/2 \Rightarrow \{m = 2 \times s\} n^2 = (2 \times s)^2/2 = 2 \times s^2\}$
$(m = 2 \times s), (n^2 = 2 \times s^2), (m, n \text{ have no common factor})$
$\Rightarrow \{\text{from } (m = 2 \times s), (m \text{ is even}); \text{from } n^2 = 2 \times s^2, n \text{ is even}\}$
$(m \text{ is even}), (n \text{ is even}), (m, n \text{ have no common factor})$
$\Rightarrow \{\text{since } m, n \text{ are both even, they have a common factor, namely } 2\}$
<i>false</i>

In order to show $true \Rightarrow \sqrt{2}$ is irrational, I showed $\sqrt{2}$ rational $\Rightarrow false$; that is, I proved $p \Rightarrow q$ by proving its contrapositive $\neg q \Rightarrow \neg p$. In a proof of $p \Rightarrow q$ by contradiction, we can also use the rule that $p \Rightarrow q$ is equivalent to $(p \wedge \neg q) \Rightarrow false$; that is, derive a contradiction by assuming that the hypothesis holds and the conclusion does not.

2.8.1.2 Pairing Points with Non-intersecting Lines

I show a result from plane geometry: given an even number of points in a plane where no three points are on a line, it is possible to find a pairing of the points so that the line segments connecting the pairs do not intersect.

Let pairing P have the minimum combined length of the line segments over all pairings. I show that P is intersection-free. The proof is by contradiction.

Suppose line segments (a, b) and (c, d) in P intersect, as shown by solid lines in Figure 2.11(a). The point of intersection z is distinct from any of the points a, b, c and d because no three points are on a line. Redirect the pairing to (a, c) and (b, d) , as shown by solid lines in Figure 2.11(b). Using $|pq|$ to denote the length of line segment (p, q) :

$$\begin{aligned}
 & |ac| + |bd| \\
 < & \{\text{triangle inequality: } |ac| < |az| + |zc|, |bd| < |bz| + |zd|\} \\
 & |az| + |zc| + |bz| + |zd| \\
 = & \{\text{rewriting}\} \\
 & |az| + |bz| + |zc| + |zd| \\
 = & \{\text{geometry}\} \\
 & |ab| + |cd|
 \end{aligned}$$

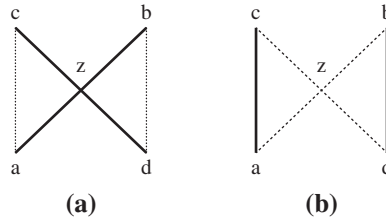


Figure 2.11 Crossing lines ab and cd in (a) are paired as ac and bd in (b).

Thus, the new pairing has smaller combined length of its line segments than P , one having the minimum combined length of the line segments over all pairings, a contradiction.

2.8.1.3 Euclid's Proof of Infinity of Primes, by Contradiction

Euclid, perhaps 2,500 years ago, proved that there is an infinite number of primes, using contradiction. Suppose there is a finite set of primes, P . Take the product of all elements of P and add 1 to it to get a number p . I show that p is greater than 1, a prime and a composite number; a contradiction.

Product of the elements of any set, even the empty set, is at least 1; so p is at least 2. From the construction, p is not divisible by any element of P ; so, it is a prime. And, p is larger than any element of P ; so, it does not belong to P , and hence not a prime.

2.8.2 Constructive and Non-constructive Existence Proofs

I show examples of a constructive and a non-constructive existence proof.

A constructive proof Show that for every positive integer n , there are n consecutive positive integers that are all composites. For $n = 2$, we have $\langle 8, 9 \rangle$; for $n = 3$, the sequence $\langle 8, 9, 10 \rangle$ works, and for $n = 5$ take $\langle 24, 25, 26, 27, 28 \rangle$. In general, for any n let $x = (n + 1)! + 1$. The n consecutive integers $\langle x + 1, \dots, x + i, \dots, x + n \rangle$ are all composites. This is because for each i , $x + i = (n + 1)! + (i + 1)$ and both of its terms are divisible by $i + 1$, so it is a composite.

A non-constructive proof Show that there are irrationals a and b such that a^b is rational.

1. Suppose $\sqrt{2}^{\sqrt{2}}$ is rational: Then $a, b = \sqrt{2}, \sqrt{2}$.
2. Suppose $\sqrt{2}^{\sqrt{2}}$ is irrational: Then $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2} \times \sqrt{2}} = \sqrt{2}^2 = 2$. In this case let $a, b = \sqrt{2}^{\sqrt{2}}, \sqrt{2}$.

The proof uses the law that for any p and q , if $p \Rightarrow q$ and $\neg p \Rightarrow q$, then q holds. This law is known as the “law of excluded middle”. Here p is “ $\sqrt{2}^{\sqrt{2}}$ is rational” and q is “there are irrationals a and b such that a^b is rational”.

2.8.3 Sum and Product Puzzle

I show a somewhat involved example of translating a problem statement to formal logic. It is given that there are two unknown integers, u and v , both of which are greater than 1 and whose sum does not exceed 100. There are two parties, S and P , where P has been given the product $u \times v$, and S the sum $u + v$. They engage in the following conversation.

1. First S asserts: P does not know the values of u and v .
2. Then P asserts: I now know the values of u and v .
3. Then S asserts: I now know the values of u and v .

We are required to determine the pair (u, v) . The description of the puzzle implies that there is a unique pair that satisfies the claims in this conversation.

This is an excellent example of a problem that combines reasoning with computation. I emphasize the reasoning part in the following development: convert the given information to a predicate such that the pair (u, v) is the unique witness to its satisfiability.

Solution I will write a number of predicates over the unknowns u and v so that the witness is a pair (u, v) where $u \leq v$.

1. The initial knowledge is:

$$b_0(u, v) :: 1 < u, u \leq v, u + v \leq 100.$$

2. In the first step S asserts that P does not know the values of u and v . Since S only knows that $b_0(u, v)$ holds and the value of $u + v$, she must have deduced that for *every* possible split of the sum $u + v$ into x and y so that $u + v = x + y$ and $b_0(x, y)$ holds, and the product $x \times y$ can not be uniquely factored into two factors. Therefore, no matter what the values of u and v are, it is not possible to deduce u and v given only $u \times v$. Write $nuf(z)$ to denote that positive integer z does not have unique factorization into two factors. I show a simpler way of writing nuf afterwards. I encode the first assertion of S by:

$$b_1(u, v) :: b_0(u, v), (\forall x, y : b_0(x, y), x + y = u + v : nuf(x \times y)).$$

3. In the second step P asserts that P knows the values of u and v . This means that it is possible to compute unique values of u and v from b_1 and $u \times v$. That is,

$$b_2(u, v) :: \{(x, y) \mid b_1(x, y), x \times y = u \times v\} = \{(u, v)\}.$$

There are several pairs (u, v) that satisfy b_2 , such as $(2, 9)$ and $(2, 25)$. But P can uniquely identify the pair from his knowledge of $u \times v$. We can not because we do not know $u \times v$.

Observe that the definition of $b_2(u, v)$ requires that the set

$$\{(x, y) \mid b_1(x, y), x \times y = u \times v\}$$

be exactly $\{(u, v)\}$; it does not say that the size of the set is 1. Such a formulation would admit the possibility that some pair of values (r, s) , different from (u, v) , satisfies b_1 and $r \times s = u \times v$.

4. In the third step S asserts that S knows the values of u and v . This means that for all possible splits of $u + v$ into x and y , i.e. $u + v = x + y$, there is just one pair, (u, v) , for which $b_2(x, y)$ holds:

$$b_3(u, v) :: \{(x, y) \mid b_2(x, y), x + y = u + v\} = \{(u, v)\}.$$

The puzzle claims that there is a unique witness to the satisfiability of $b_3(u, v)$. The computation of the witness requires enumeration of pairs which can be minimized using several shortcuts; see, for example, transcription of a note in [Dijkstra \[2003\]](#). I show one simplification below. The unique pair is (4, 13). Vladimir Lifschitz has translated a similar formulation to a problem in answer-set programming. The solver, *clingo*, outputs (4, 13) as the unique witness, to no one's great surprise.

Simplifying predicate b_1 I show a simpler way to write predicate *nuf* in b_1 . It is easily shown that a positive integer z can be uniquely factored into two factors iff z is either of the form (1) $p \times q$ for distinct primes p and q , so the factors are (p, q) , or (2) p^3 for a prime p , so the factors are (p, p^2) . (Published solutions often miss the second case.)

I will use the fact that every even integer greater than 2 and at most 100 can be written as a sum of two primes. This is a version of Goldbach's conjecture which claims this result for all even integers greater than 2. Though that conjecture still remains a conjecture, the result is definitely true for integers less than or equal to 100.

If $u + v$ is an even number, from Goldbach's conjecture, it can be split into two prime numbers; so we deduce that $u + v$ is odd. If $(u, v) = (p, p^2)$ for some prime p , then $u + v$ is even because $p + p^2 = p \cdot (p + 1)$; so this possibility is denied by the condition that $u + v$ is odd. Additionally, if $u + v - 2$ is prime then $u + v$ can be split to $(2, u + v - 2)$ where both components are prime. So, $u + v - 2$ has to be a composite number. Therefore,

$$b_1(u, v) :: b_0(u, v), \text{ odd}(u + v), \text{ composite}(u + v - 2),$$

where the predicates *odd* and *composite* have their usual meanings.

2.8.4 Russell's Paradox

Gottlob Frege, the founder of modern logic, wrote two volumes of an influential book, *The Foundations of Arithmetic*. In this book, he formulated the set comprehension principle that is given in Section 2.1.3 (page 10): for any predicate P there is a unique set defined by $\{x \mid P(x)\}$. Just before publication of the second volume of the book, Bertrand Russell showed that this principle leads to a contradiction (see [van Heijenoort \[2002\]](#) for a complete discussion). Here is part of the letter that Russell wrote to Frege:

The comprehensive class we are considering, which is to embrace everything, must embrace itself as one of its members. In other words, if there is such a thing as "everything", then, "everything" is something, and is a

member of the class "everything". But normally a class is not a member of itself. Mankind, for example, is not a man. Form now the assemblage of all classes which are not members of themselves. This is a class: is it a member of itself or not? If it is, it is one of those classes that are not members of themselves, i.e. it is not a member of itself. If it is not, it is not one of those classes that are not members of themselves, i.e. it is a member of itself. Thus of the two hypotheses—that it is, and that it is not, a member of itself—each implies its contradictory. This is a contradiction.

Modern notation can simplify this informal argument substantially. Define set S by comprehension: $S = \{x \mid x \notin x\}$. That is, for any z , $z \in S \equiv z \notin z$. Now, derive a contradiction.

$$\begin{aligned}
 & (\forall z :: z \in S \equiv z \notin z) \\
 \Rightarrow & \{\text{instantiate } z \text{ by } S\} \\
 & S \in S \equiv S \notin S \\
 \equiv & \{(p \equiv \neg p) \equiv \text{false, for any predicate } p. \text{ Let } p \text{ be } S \in S.\} \\
 & \text{false}
 \end{aligned}$$

In view of this result, set comprehension has to be used with care, as I have described in Section 2.1.3 (page 10). The purpose of this example is to illustrate that since Russell's time we have learned much about proof presentation, so that a formal proof is simpler to comprehend than a wordy explanation.

2.8.5 Cantor's Diagonalization

The powerset of a finite set of n elements, $n \geq 0$, has 2^n elements (see Section 2.1.4). Therefore, the powerset of a finite set is larger in size than the set itself. Cantor showed an equivalent result for infinite sets. The difficulty in comparing the sizes of infinite sets is that we cannot count the number of elements; so, a different strategy has to be employed.

Cantor defined two sets to be equal in size iff their elements can be paired, one-to-one.⁶ Define $|S| \leq |T|$ if every element of set S can be mapped to a unique element of set T , that is, if there is an injective function from S to T , $|S| \geq |T|$ if there is a surjective function from S to T , and $|S| = |T|$ if there is a bijective function from S to T . Consequently, if there is an injective function from S to T but no bijective

6. Actually, this observation is due to Leibniz. He showed that the set of natural numbers has the same size as the set of even natural numbers. But he abandoned defining the cardinalities of such sets because "a part cannot be as large as the whole".

function, then $|S| < |T|$; this is how one infinite set is shown to be smaller than another.⁷

Cantor proved that every set is strictly smaller than its powerset. That is, there is no way to pair up the elements of a set with the elements of its powerset. There is a beautiful proof of this result for the set of natural numbers $\mathbb{N}$ using *diagonalization* (see Exercise 20, page 81). I prove the more general result for arbitrary infinite sets here.

Cantor's theorem in popular terms Gardner [2001, pp. 340–341] includes a proof that he calls “one of the most beautiful proofs in set theory”. The proof is suitable for a general audience since it uses no mathematical symbols at all; I reproduce the proof verbatim.

It is an indirect proof, a *reductio ad absurdum*. Assume that all elements of N , a set with any number or members, finite or infinite, are matched one-to-one with all of N 's subsets. Each matching defines a coloring of the elements:

1. An element is paired with a subset that includes that element. Let us call all such elements blue.
2. An element is paired with a subset that does not include that element. We call all such elements red.

The red elements form a subset of our initial set N . Can this subset be matched to a blue element? No, because every blue element is in its matching subset, therefore the red subset would have to include a blue element. Can the red subset be paired with a red element? No, because the red element would then be included in its subset and would therefore be blue. Since the red subset cannot be matched to either a red or blue element of N , we have constructed a subset of N that is not paired with any element of N . No set, even if infinite, can be put into one-to-one correspondence with its subsets.

For a reader who does not have training in formal algebra and logic, or who does not believe in formalism, this is undoubtedly an excellent proof. But is this the simplest proof for a reader who has some formal training and is not shy about using it? I find this proof unsatisfactory because it requires more mental effort than it deserves. I show a proof I believe to be simpler [see Dijkstra and Misra 2001].

7. If $|S| \leq |T|$, there is an injective function from S to T , and if $|T| \leq |S|$, there is an injective function from T to S . If $|S| \leq |T|$ and $|T| \leq |S|$, using the Cantor–Schröder–Bernstein theorem (see Section 2.2.1.4, page 20), there is a bijection between S and T ; so $|S| = |T|$.

Cantor's theorem in formal terms Let S be a set, finite or infinite, and T its powerset. There is an injective function from S to T , the function that maps element x of S to $\{x\}$. So, $|S| \leq |T|$. I show that there is no bijection between S and T ; the proof is by contradiction.

Suppose there exists a bijective function $f, f: S \rightarrow T$. Define set R by:

$$(z \in R) \equiv z \notin f(z). \quad (\text{DM})$$

$$\begin{aligned} & \text{true} \\ \Rightarrow & \{R \subseteq S. \text{ And } T \text{ is the powerset of } S.\} \\ & R \in T \\ \Rightarrow & \{f: S \rightarrow T \text{ is bijective. So it has an inverse, } f^{-1}: T \rightarrow S\} \\ & f^{-1}(R) \in S \\ \Rightarrow & \{\text{instantiate (DM) with } f^{-1}(R) \text{ for } z\} \\ & (f^{-1}(R) \in R) \equiv (f^{-1}(R) \notin f(f^{-1}(R))) \\ \equiv & \{f(f^{-1}(R)) = R\} \\ & (f^{-1}(R) \in R) \equiv (f^{-1}(R) \notin R) \\ \equiv & \{\text{predicate calculus}\} \\ & \text{false} \end{aligned}$$

The set R in this proof is indeed the set of red elements in the informal proof. I did not need the concept of blue (which is implicit), nor is there a need for case analysis about the color of the element with which R is paired. This simplification is the gift of formalism.

Countably infinite set Any set that can be put in 1–1 correspondence with the set of natural numbers is called *countably infinite*. Each of its elements can be indexed by a natural number. It can be shown that all of the following sets are countably infinite: (1) set of all integers, (2) set of all positive integers, (3) set of rational numbers, (4) union of countable number of countably infinite sets, and (5) set of finite sequences of elements from a countably infinite set (see Exercise 21, page 82).

From Cantor's proof, the set of real numbers is not countably infinite. In fact, the real interval between 0 and 1 is not countably infinite.

2.8.6 Saddle Point

An element of a matrix of numbers is a *saddle point* if it is the largest in its row *and* the smallest in its column. In Table 2.5 the bottom left entry, shown in boldface, is a saddle point. The reader may verify that there are no other saddle points.

A matrix may have multiple saddle points, for instance when all its elements are equal, or it may have no saddle point, as in a 2×2 matrix whose diagonal elements are 0 and off-diagonal ones are 1. I develop the condition for the existence

Table 2.5 A matrix with a saddle point, shown in bold

9	2	6
7	4	0
5	3	1

of a saddle point and derive an algorithm to locate one if it exists. This example illustrates the usefulness of our chosen notation and proof style. For a matrix A define

$$\begin{aligned} hi(i) &= \text{the largest value in row } i, & \text{i.e., } hi(i) &= (\max j :: A[i, j]) \\ lo(j) &= \text{the smallest value in column } j, & \text{i.e., } lo(j) &= (\min i :: A[i, j]). \end{aligned}$$

$A[u, v]$ is called a saddle point iff $lo[v] = A[u, v] = hi(u)$.

Proposition 2.1 $A[u, v]$ is a saddle point iff $(\max j :: lo(j)) = A[u, v] = (\min i :: hi(i))$.

Proof.

$$\begin{aligned} & A[u, v] \text{ is a saddle point} \\ \equiv & \{ \text{definition of saddle point} \} \\ & lo(v) = A[u, v] = hi(u) \\ \equiv & \{ \text{from the definition of } lo \text{ and } hi: (\forall i, j :: lo(j) \leq A[i, j] \leq hi(i)); \\ & \text{so } (\forall i, j :: lo(j) \leq hi(i)), \text{ and } (\max j :: lo(j)) \leq (\min i :: hi(i)) \} \\ & lo(v) = A[u, v] = hi(u), \quad (\max j :: lo(j)) \leq (\min i :: hi(i)) \\ \equiv & \{ \text{rewriting} \} \\ & lo(v) = A[u, v] = hi(u), \quad lo(v) \leq (\max j :: lo(j)) \leq (\min i :: hi(i)) \leq hi(u) \\ \equiv & \{ \text{arithmetic} \} \\ & lo(v) = A[u, v] = hi(u), \quad lo(v) = (\max j :: lo(j)) = (\min i :: hi(i)) = hi(u) \\ \equiv & \{ \text{arithmetic} \} \\ & (\max j :: lo(j)) = A[u, v] = (\min i :: hi(i)) \end{aligned} \quad \blacksquare$$

We now have an algorithm to detect if a matrix has a saddle point: compute the largest element of each row and the smallest of each column; check if the smallest among the former is equal to the largest among the latter. In Table 2.6, I have computed the hi and lo values for the matrix in Table 2.5. As expected, $(\max j :: lo(j)) = 5 = (\min i :: hi(i))$.

Table 2.6 hi and lo values for the matrix in Table 2.5

9	2	6	9
7	4	0	7
5	3	1	5
5	2	0	

2.8.7 Properties of the Least Upper Bound of Partial Order

In Section 2.4.2.2 (page 35), I defined the least upper bound ($\text{lub}, \uparrow$) and the greatest lower bound ($\text{glb}, \downarrow$) of a partial order $(S, \leq)$. I derive certain properties of $\uparrow$: it is commutative, associative, idempotent and monotonic. Similar properties also hold for $\downarrow$. First, I prove two general results about partial orders that are of interest in their own rights.

Below, $\uparrow$ has precedence over $\leq$, so $x \uparrow y \leq z$ means $(x \uparrow y) \leq z$.

Proposition 2.2 (Indirect proof of ordering) For any partial order $\leq$,
 $(y \leq x) \equiv (\forall w :: x \leq w \Rightarrow y \leq w)$, for all x and y .

Proof. Proof is by mutual implication.

- $(y \leq x) \Rightarrow (\forall w :: x \leq w \Rightarrow y \leq w)$: Assume $y \leq x$.

$$\begin{aligned}
 & x \leq w \\
 \Rightarrow & \text{\{given that } y \leq x\}} \\
 & y \leq x \wedge x \leq w \\
 \Rightarrow & \text{\{\leq is a partial order, hence transitive\}} \\
 & y \leq w
 \end{aligned}$$

- $(\forall w :: x \leq w \Rightarrow y \leq w) \Rightarrow (y \leq x)$:

$$\begin{aligned}
 & (\forall w :: x \leq w \Rightarrow y \leq w) \\
 \Rightarrow & \text{\{instantiate } w \text{ by } x\}} \\
 & x \leq x \Rightarrow y \leq x \\
 \Rightarrow & \text{\{predicate calculus: } x \leq x \equiv \text{true} \text{ and } (\text{true} \Rightarrow p) \equiv p\}} \\
 & y \leq x
 \end{aligned}$$

■

Proposition 2.3 Indirect proof of equality

For any partial order $\leq$,

$$(x = y) \equiv (\forall w :: x \leq w \equiv y \leq w), \text{ for all } x \text{ and } y.$$

Proof. Using Proposition 2.1, $(\forall w :: x \leq w \Rightarrow y \leq w) \equiv (y \leq x)$. And instantiating x by y and y by x in Proposition 2.1, $(\forall w :: y \leq w \Rightarrow x \leq w) \equiv (x \leq y)$. The result follows. ■

Proposition 2.4 gives an alternate characterization of the least upper bound that is more amenable to formal manipulation.

Proposition 2.4 $x \uparrow y \leq z \equiv x \leq z \wedge y \leq z$, for all x, y and z .

Proof.

$$\begin{aligned}
 & x \leq z \wedge y \leq z \\
 \equiv & \{\text{Proposition 2.2}\} \\
 & (\forall w :: z \leq w \Rightarrow x \leq w) \wedge (\forall w :: z \leq w \Rightarrow y \leq w) \\
 \equiv & \{\text{rewrite}\} \\
 & (\forall w :: z \leq w \Rightarrow (x \leq w \wedge y \leq w)) \\
 \equiv & \{\text{definition of } \uparrow\} \\
 & (\forall w :: z \leq w \Rightarrow x \uparrow y \leq w) \\
 \equiv & \{\text{Proposition 2.2}\} \\
 & x \uparrow y \leq z
 \end{aligned}$$

Proposition 2.5 $\uparrow$ is commutative.

Proof. For any x, y, w

$$\begin{aligned}
 & x \uparrow y \leq w \\
 \equiv & \{\text{Proposition 2.4}\} \\
 & x \leq w \wedge y \leq w \\
 \equiv & \{\text{Commutativity of } \wedge\} \\
 & y \leq w \wedge x \leq w \\
 \equiv & \{\text{Proposition 2.4}\} \\
 & y \uparrow x \leq w
 \end{aligned}$$

Using Proposition 2.3 on $(\forall w :: x \uparrow y \leq w \equiv y \uparrow x \leq w)$, conclude $(x \uparrow y) \equiv (y \uparrow x)$.

Proposition 2.6 $\uparrow$ is associative.

Proof. For any x, y, z, w ,

$$\begin{aligned}
 & (x \uparrow y) \uparrow z \leq w \\
 \equiv & \{\text{Proposition 2.4}\} \\
 & (x \uparrow y) \leq w \wedge z \leq w \\
 \equiv & \{\text{Proposition 2.4}\} \\
 & (x \leq w \wedge y \leq w) \wedge z \leq w \\
 \equiv & \{\text{Associativity of } \wedge\} \\
 & x \leq w \wedge (y \leq w \wedge z \leq w) \\
 \equiv & \{\text{Proposition 2.4}\} \\
 & x \leq w \wedge (y \uparrow z) \leq w \\
 \equiv & \{\text{Proposition 2.4}\} \\
 & x \uparrow (y \uparrow z) \leq w
 \end{aligned}$$

Using Proposition 2.3, $(x \uparrow y) \uparrow z = x \uparrow (y \uparrow z)$.

2.8.8 Knaster–Tarski Theorem

This section contains advanced material and may be skipped on first reading.

The theorem of Knaster–Tarski [Tarski 1955] is of fundamental importance in set theory, algebra and logic. It has numerous applications in program semantics. A simple proof of the Cantor–Schröder–Bernstein theorem in Section 2.2.1.4 (page 20) follows from this theorem. The theorem gives a complete characterization of the solutions of the equation $f(x) = x$ where f is a monotonic function on a *complete lattice*, which is defined below.

The theorem can be appreciated with even a rudimentary knowledge of set theory and logic. I describe just enough of lattice theory to state and prove the Knaster–Tarski theorem. I give the proof in detail because I want the reader to appreciate the structure of the proof, given in small steps along with their justifications.

Complete lattice A *lattice* is a partially ordered set $(L, \leq)$ such that every pair of elements of L has a greatest lower bound (glb) and a least upper bound (lub) in L . And $(L, \leq)$ is a *complete lattice* if every subset of L has a greatest lower bound and a least upper bound in L . Recall from Section 2.4.2.2 (page 35) that $\text{lub}(S)$ and $\text{glb}(S)$ denote, respectively, the lub and glb of set S . Function f is monotonic over $(L, \leq)$ if for every x and y , where $x \leq y$, $f(x) \leq f(y)$.

A complete lattice $(L, \leq)$ is never empty because it includes its own glb and lub, $\perp_L$ and $\top_L$, respectively. I show $\text{lub}(\{\}) = \perp_L$ and $\text{glb}(\{\}) = \top_L$. The proof for $\text{lub}(\{\})$ is as follows, and for $\text{glb}(\{\})$ analogous:

- (1) Every element of L is an upper bound of $\{\}$ from $(\forall x, y : x \in \{\}, y \in L : x \leq y)$, and
- (2) $\perp_L$ is the least upper bound from $(\forall y : y \in L : \perp_L \leq y)$.

Statement of the Knaster–Tarski theorem A fixed point of function f satisfies $f(x) = x$. The *least* fixed point x is a fixed point, and for any fixed point y , $x \leq y$. Similarly, the *greatest* fixed point z is a fixed point, and for any fixed point y , $y \leq z$.

Theorem 2.1 Knaster–Tarski

For a monotonic function f over a complete lattice $(L, \leq)$:

1. the least and the greatest fixed points of f exist, and
2. the fixed points of f form a complete lattice. ■

In most applications in computer science only part (1) of the theorem is used. Part (2) of the theorem is a stronger statement; it implies part (1) as follows. Let F be the set of fixed points of f . From (2) F is a complete lattice, so $\text{glb}(F)$ and $\text{lub}(F)$ are in F , and they are the least and the greatest fixed points of f , respectively.

Interval of a complete lattice is a complete lattice Recall the definition of interval from Section 2.4.2.3 (page 36). I show that any interval $I = [a, b]$ of the complete lattice $(L, \leq)$ is a complete lattice. More formally, $(I, \leq_I)$ is a complete lattice, where the order relation $\leq_I$ is same as $\leq$ but restricted to set I , so $(\forall x, y : x \in I, y \in I : x \leq_I y \equiv x \leq y)$.

I use the following elementary facts about lub and glb in the proof:

For non-empty set S : $glb(S) \leq lub(S)$.

For $T \subseteq S$: $lub(T) \leq lub(S)$ and $glb(S) \leq glb(T)$.

Lemma 2.1 $(I, \leq_I)$ is a complete lattice where $I = [a, b]$.

Proof.

(1) $glb_I(I) = a$ and $lub_I(I) = b$:

From the definition of I , $a \in I$ and a is a lower bound of I ; so $glb_I(I) = a$. Similarly, $lub_I(I) = b$ and $b \in I$.

Next, I show that for any subset T of I , $glb_I(T) \in I$ and $lub_I(T) \in I$, which completes the proof that $(I, \leq_I)$ is a complete lattice. I prove two cases, $T = \{\}$ in (2) and $T \neq \{\}$ in (3).

(2) $lub_I(\{\}) \in I$, $glb_I(\{\}) \in I$:

$$\begin{aligned} & lub_I(\{\}) = glb_I(I), glb_I(\{\}) = lub_I(I) \\ \Rightarrow & \text{from (1)} \\ & lub_I(\{\}) \in I, glb_I(\{\}) \in I \end{aligned}$$

(3) For $T \neq \{\}$, $lub_I(T) \in I$, $glb_I(T) \in I$:

I prove only that $lub_I(T) \in I$; the proof of $glb_I(T) \in I$ is analogous. For readability, I omit the subscript I in the following proof.

$$\begin{aligned} & \text{true} \\ \Rightarrow & \{T \subseteq I, \text{ so } glb(I) \leq glb(T)\} \\ & glb(I) \leq glb(T) \\ \Rightarrow & \{T \neq \{\}, \text{ so } glb(T) \leq lub(T)\} \\ & glb(I) \leq lub(T) \\ \Rightarrow & \{T \subseteq I; \text{ so } lub(T) \leq lub(I)\} \\ & glb(I) \leq lub(T) \leq lub(I) \\ \Rightarrow & \{\text{from (1) } glb(I) = a \text{ and } lub(I) = b. \text{ Apply definition of } I\} \\ & lub(T) \in I \end{aligned}$$

■

2.8.8.1 Proof of Theorem 2.1 (part 1)

For function $f: L \rightarrow L$, element x of L is a (1) *prefixed point* if $f(x) \leq x$, and (2) *post-fixed point* if $x \leq f(x)$. Thus, a fixed point is both a prefixed and a postfixed point.

Analogous to the least and greatest fixed points, the least and greatest prefixed and postfix points are defined. Henceforth, *point* refers to an element of L .

For Theorem 2.1 (part 1), I prove a slightly stronger result, that the least fixed and the least prefixed points of f exist, and they are identical. The analogous result, that the greatest fixed and the greatest postfix points of f exist and they are identical, can be proved similarly by using lub for glb and postfix points for prefixed points.

Let pre be the set of prefixed points of f . Set pre is non-empty because $f(\top_L) \leq \top_L$, so $\top_L \in pre$. Also, pre is a subset of the complete lattice L , so it has a glb p . I show that p is both the least prefixed point and the least fixed point of f .

- p is the least prefixed point: For any prefixed point x ,

$$\begin{aligned}
 & \text{true} \\
 \Rightarrow & \{p \text{ the glb of } pre \text{ and } x \in pre\} \\
 & p \leq x \\
 \Rightarrow & \{f \text{ is monotonic}\} \\
 & f(p) \leq f(x) \\
 \Rightarrow & \{x \text{ is a prefixed point; so, } f(x) \leq x\} \\
 & f(p) \leq x \\
 \Rightarrow & \{x \text{ is an arbitrary point in } pre\} \\
 & f(p) \text{ is a lower bound of } pre \\
 \Rightarrow & \{p \text{ is the glb of } pre\} \\
 & f(p) \leq p \\
 \Rightarrow & \{\text{definition of prefixed point applied to } p\} \\
 & p \in pre \\
 \Rightarrow & \{p \text{ is the glb of } pre, \text{ definition of the least prefixed point}\} \\
 & p \text{ is the least prefixed point}
 \end{aligned}$$

- p is the least fixed point:

$$\begin{aligned}
 & \text{true} \\
 \Rightarrow & \{p \text{ is a prefixed point from the above proof}\} \\
 & f(p) \leq p \\
 \Rightarrow & \{f \text{ is monotonic}\} \\
 & f(f(p)) \leq f(p) \\
 \Rightarrow & \{\text{definition of prefixed point applied to } f(p)\}
 \end{aligned}$$

$$\begin{aligned}
& f(p) \text{ is a prefixed point} \\
\Rightarrow & \{p \text{ is the least prefixed point}\} \\
& p \leq f(p) \\
\Rightarrow & \{p \text{ is a prefixed point; so } f(p) \leq p\} \\
& p = f(p) \\
\Rightarrow & \{\text{fixed points} \subseteq \text{prefixed points; } p \text{ is the least prefixed point}\} \\
& p \text{ is the least fixed point}
\end{aligned}$$

2.8.8.2 Proof of Theorem 2.1 (part 2)

In order to show that F , the set of fixed points of function f , is a complete lattice, we have to show that any subset of F , including the empty set, has a lub and glb in F . To show the existence of the lub for any subset r of F , prove that there is a least fixed point among the upper bounds of r , and for the glb the existence of a greatest fixed point among the lower bounds of r . The two proofs are similar; so I prove the result only for lub.

Define the set of upper bounds of r by $R = \{w \mid \text{lub}(r) \leq w\}$. Observe that R is the interval $[\text{lub}(r), \top_L]$. Therefore, R is a complete lattice, from Lemma 2.1 (page 71). Now the proof is in two parts: (1) show that f maps elements of R to R (Lemma 2.2, page 73) and (2) there is a least fixed point of f in R , which is the desired lub of r in F (see Lemma 2.3, page 74).

Lemma 2.2 f maps elements of R to R .

Proof. If r is empty, $R = L$, and f maps elements of L to L . I prove the result for non-empty r . Let $x \in r$ and $y \in R$ (R is non-empty because $\text{lub}(r) \in R$):

$$\begin{aligned}
& x \in r, y \in R \\
\Rightarrow & \{x \leq \text{lub}(r); \text{lub}(r) \leq y \text{ from the definition of } R\} \\
& x \leq y \\
\Rightarrow & \{f \text{ is monotonic}\} \\
& f(x) \leq f(y) \\
\Rightarrow & \{S \text{ is a set of fixed points; so, } f(x) = x\} \\
& x \leq f(y) \\
\Rightarrow & \{x \text{ is an arbitrary element of } r; \text{ so } f(y) \text{ is an upper bound of } r\} \\
& \text{lub}(r) \leq f(y) \\
\Rightarrow & \{\text{definition of } R\} \\
& f(y) \in R
\end{aligned}$$

■

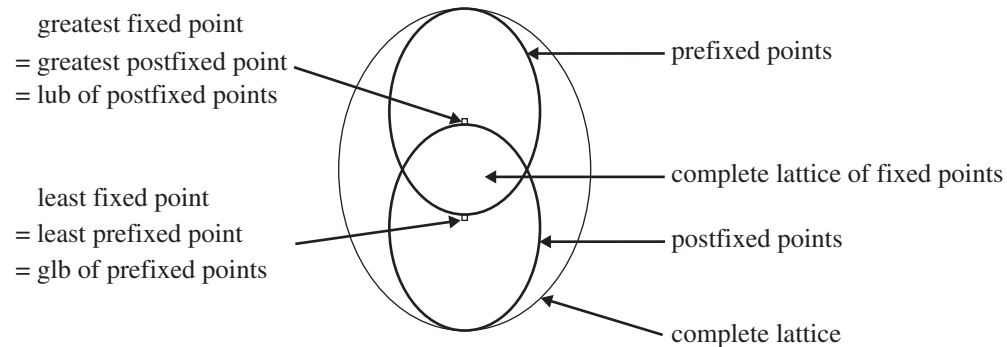


Figure 2.12 Pictorial depiction of the Knaster-Tarski theorem.

Lemma 2.3 $\text{lub}_F(r) \in F$.

Proof.

R is a complete lattice, f is a monotone mapping over R
 $\Rightarrow$ {from Theorem 2.1 (part 1), page 70}
 f has a least fixed point q in R
 $\Rightarrow$ $\{q \text{ is an upper bound of } r \text{ since } q \in R\}$
 $q = \text{lub}_F(r)$ ■

A pictorial depiction of the Knaster-Tarski theorem is shown in Figure 2.12.

2.9 Exercises

- What is the set of divisors of $p \cdot q$ where p and q are distinct primes?
Solution $\{1, p, q, p \cdot q\}$.
- Prove that symmetric difference of sets is an associative operation.
- Is the successor function on the set of naturals bijective?
- Let $h: R \rightarrow Z$, where R is the set of reals, Z the set of integers and $h(x)$ the largest integer not exceeding x . Is h surjective?
- Show functions that are: (1) neither injective nor surjective, (2) injective but not surjective, (3) surjective but not injective.
- Let 0_S and id_S be the vacuous and identity relations, respectively, over set S , and R any binary relation over S . Recall that “;” is used for composition of relations. Show that (1) $R \cup 0_S = R$, (2) $(R; 0_S) = (0_S; R) = 0_S$, and (3) $(id_S; R) = (R; id_S) = R$.
- Show relations that are reflexive and symmetric but not transitive, reflexive and transitive but not symmetric, and symmetric and transitive but not reflexive.

Partial solution I show a relation that is symmetric and transitive but not reflexive. Define R over the set of integers where xRy if $x \times y$ is odd. Then xRy holds iff both x and y are odd. The relation is symmetric. It is transitive, by the same argument. However, it is not reflexive: xRx does not hold for any even x .

8. I have shown in Example 2.8 (page 57) that the intersection of two equivalence relations is an equivalence relation. Are analogous results true for union and composition? Is the complement of an equivalence relation an equivalence relation?

Partial solution Composition of two equivalence relations is not necessarily an equivalence relation. Consider integers 1 through 5 as the domain of the relations. Let equivalence relations R and R' be given by:

$$x R y \text{ iff } \lceil x/2 \rceil = \lceil y/2 \rceil$$

$$x R' y \text{ iff } \lfloor x/2 \rfloor = \lfloor y/2 \rfloor$$

where R and R' use the ceiling and floor functions, respectively. Then $1 R 2$ and $2 R' 3$, so $1 (R; R') 3$. From $3 R 4$ and $4 R' 5$, so $3 (R; R') 5$. However, there is no x such that $1 R x$ and $x R' 5$, so $1 (R; R') 5$ does not hold. Hence, $(R; R')$ is not transitive, so it is not an equivalence relation.

9. Prove that the composition of two equivalence relations over the same set is an equivalence relation if and only if their composition commutes, that is, composition in either order gives the same relation.
10. For infinite binary strings x and y , write xRy if x and y differ in only a finite number of corresponding positions (i.e., $x_i = y_i$, for almost all i). Show that R is an equivalence relation. Show that the number of equivalence classes is infinite.

Partial solution I show that the number of equivalence classes is infinite. Let z_i , $i \geq 1$, be a string that has infinite number of alternating blocks of i zeroes followed by i ones. I show that z_i and z_j , where $j < i$, differ in an infinite number of positions. Pick a block of length i in z_i that is either all zeroes or all ones. The corresponding positions in z_j cannot be all zeroes or all ones because there are no more than j identical bits starting at any position in z_j and $j < i$. So, each block of z_i differs in a positive number of positions from that of z_j . Since there are an infinite number of blocks of z_i , z_i and z_j differ in an infinite number of positions, so they belong to distinct equivalence classes. Therefore, each z_i , $i \geq 1$, is in a different equivalence class.

11. Consider the “not equal” ($\neq$) relation over a set of at least one element. What are its (1) reflexive, (2) symmetric and (3) transitive closure?

Solution Denote the reflexive closure by $\neq'$. For distinct x and y , by definition, $x \neq y$, so $x \neq' y$. And $x \neq' x$ for all x because $\neq'$ is reflexive. So, $\neq'$ holds for every pair of elements, which may or may not be distinct.

The symmetric closure is the $\neq$ relation itself.

The transitive closure over a set of one element is the empty relation, as is the $\neq$ relation itself. For a set with more than one element the transitive closure is the universal relation, because for any two distinct elements $x \neq^+ y$ by definition, and $x \neq y \wedge y \neq x \Rightarrow x \neq^+ x$; so $x \neq^+ y$ for all x and y .

12. An *extension* of a partial order R is a partial order R' such that as sets $R \subseteq R'$; that is, if xRy then $xR'y$.

Let R' be an extension of partial order R . Show that both R and R' are extensions of the partial order $R \cap R'$, and $R \cap R'$ is the largest relation having this property. Note that, trivially, both R and R' are extensions of the empty relation. Dually, show that $R \cup R'$ is the smallest extension of both R and R' .

13. For a partially ordered set with order relation $\leq$ element x is *minimal* if no element is smaller than it. Call x *least* if all elements are at least as large, that is, $x \leq y$ for all y . Show that minimal and least are different concepts and give examples of both. Show that every finite partially ordered set has a minimal element, though not necessarily a least element.

14. This exercise uses notation and terminology given for the resolution principle in Section 2.5.4.2 (page 47). For any proposition P , it is known that P^* is closed. Show that $P^*[x := v]$ is closed for any variable x and truth value v .

Solution For specificity, let v be *true*. Consider two clauses $(y \vee q)$ and $(\neg y \vee r)$ in $P^*[x := \text{true}]$ that include complementary literals, y and $\neg y$. I show that their resolvent, $(q \vee r)$, is in $P^*[x := \text{true}]$; so, $P^*[x := \text{true}]$ is closed.

Since $(y \vee q)$ is in $P^*[x := \text{true}]$, either $(y \vee q)$ or $(\neg x \vee y \vee q)$ is in P^* . Similarly, either $(\neg y \vee r)$ or $(\neg x \vee \neg y \vee r)$ is in P^* . In all cases, their resolvent, $(q \vee r)$ or $(\neg x \vee q \vee r)$, is in P^* . In both cases, $(q \vee r)[x := \text{true}]$ and $(\neg x \vee q \vee r)[x := \text{true}]$ are $(q \vee r)$, so this clause is in $P^*[x := \text{true}]$.

15. For booleans p, b, x, y, z , it is given that

$x = p \wedge q$, $y = \neg p \wedge \neg q$, $z = ((x \vee y) \equiv q)$. Express z as a simple function of p and q .

Solution

$$\begin{aligned}
& z \\
&= \{\text{given}\} \\
&\quad (x \vee y) \equiv q \\
&= \{x = (p \wedge q), y = (\neg p \wedge \neg q)\} \\
&\quad ((p \wedge q) \vee (\neg p \wedge \neg q)) \equiv q \\
&= \{\text{simplify the first operand of } \equiv\} \\
&\quad (p \equiv q) \equiv q \\
&= \{\text{rearrange terms}\} \\
&\quad p \equiv (q \equiv q) \\
&= \{(q \equiv q) = \text{true}\} \\
&\quad p \equiv \text{true} \\
&= \{\text{property of } \equiv\} \\
&\quad p
\end{aligned}$$

16. Consider proposition (1) below.

$$(p \wedge b \Rightarrow r) \wedge (p \wedge \neg b \Rightarrow s). \quad (1)$$

Call proposition q a *solution* of (1) if (1) is satisfied by substituting q for p . Show that $(r \wedge b) \vee (s \wedge \neg b)$ is the weakest solution. That is, $(r \wedge b) \vee (s \wedge \neg b)$ is a solution of (1), and if q satisfies (1), then $q \Rightarrow ((r \wedge b) \vee (s \wedge \neg b))$. Note that the strongest solution is *false*.

Solution I first show that $(r \wedge b) \vee (s \wedge \neg b)$ is a solution. Substituting $(r \wedge b) \vee (s \wedge \neg b)$ for p in the first conjunct of the predicate:

$$\begin{aligned}
& (((r \wedge b) \vee (s \wedge \neg b)) \wedge b) \Rightarrow r \\
&\equiv ((r \wedge b \wedge b) \vee (s \wedge \neg b \wedge b)) \Rightarrow r \\
&\equiv (r \wedge b) \Rightarrow r \\
&\Rightarrow \text{true}
\end{aligned}$$

Substituting $(r \wedge b) \vee (s \wedge \neg b)$ for p in the second conjunct of the predicate:

$$\begin{aligned}
& (((r \wedge b) \vee (s \wedge \neg b)) \wedge \neg b) \Rightarrow s \\
&\equiv ((r \wedge b \wedge \neg b) \vee (s \wedge \neg b \wedge \neg b)) \Rightarrow s \\
&\equiv (s \wedge \neg b) \Rightarrow s \\
&\Rightarrow \text{true}
\end{aligned}$$

Next, I show that $(r \wedge b) \vee (s \wedge \neg b)$ is the weakest solution. That is, for any solution q , $q \Rightarrow ((r \wedge b) \vee (s \wedge \neg b))$. So, we have to show

$$((q \wedge b) \Rightarrow r) \wedge ((q \wedge \neg b) \Rightarrow s) \Rightarrow (q \Rightarrow ((r \wedge b) \vee (s \wedge \neg b))).$$

The proof is:

$$\begin{aligned}
 & ((q \wedge b) \Rightarrow r) \wedge ((q \wedge \neg b) \Rightarrow s) \\
 \Rightarrow & \{ \text{from } (q \wedge b) \Rightarrow r \text{ and } (q \wedge b) \Rightarrow b, \text{ conclude } q \wedge b \Rightarrow r \wedge b \}. \\
 & \text{similarly, } (q \wedge \neg b \Rightarrow s \wedge \neg b) \} \\
 & (q \wedge b \Rightarrow r \wedge b) \wedge (q \wedge \neg b \Rightarrow s \wedge \neg b) \\
 \Rightarrow & \{ \text{property of implication: if } u \Rightarrow v \text{ and } u' \Rightarrow v' \text{ then} \\
 & (u \vee u') \Rightarrow (v \vee v'). \text{ Use } u, u', v, v' = q \wedge b, r \wedge b, q \wedge \neg b, s \wedge \neg b \} \\
 & ((q \wedge b) \vee (q \wedge \neg b)) \Rightarrow ((r \wedge b) \vee (s \wedge \neg b)) \\
 \equiv & \{ ((q \wedge b) \vee (q \wedge \neg b)) \equiv q \} \\
 & q \Rightarrow ((r \wedge b) \vee (s \wedge \neg b))
 \end{aligned}$$

17. (a) Show that if f is a dual of g , then g is a dual of f .
 (b) Show that if f and f' are duals of g , then $f = f'$.
 (c) Show that *true* and *false* are duals, $\neg$ and the identity function *id* over one argument are self-duals, and the following pairs of operators are duals:
 $(\wedge, \vee), \quad (=, \neq), \quad (\equiv, \not\equiv)$.
 (d) What is the dual of $\Rightarrow$?

Solution

- (a) *true*
 $\Rightarrow \{f \text{ is a dual of } g\}$
 $f(\neg X) = \neg g(X)$
 $\Rightarrow \{\text{negate both sides}\}$
 $\neg f(\neg X) = g(X)$
 $\Rightarrow \{\text{replace variables } X \text{ by their negations}\}$
 $\neg f(X) = g(\neg X)$
 $\Rightarrow \{\text{definition of dual}\}$
 $g \text{ is a dual of } f$
- (b) *true*
 $\Rightarrow \{f \text{ is a dual of } g \text{ and } f' \text{ is a dual of } g\}$
 $f(\neg X) = \neg g(X), f'(\neg X) = \neg g(X)$
 $\Rightarrow \{\text{property of equality}\}$
 $f(\neg X) = f'(\neg X)$
 $\Rightarrow \{\text{definition of function equality}\}$
 $f = f'$
- (c) Apply the definition of dual.

(d) Writing $\leftrightarrow$ for the dual of $\Rightarrow$, we have

$$\begin{aligned}
 & x \leftrightarrow y \\
 \equiv & \text{\{definition of dual\}} \\
 & \neg(\neg x \Rightarrow \neg y) \\
 \equiv & \{(\neg x \Rightarrow \neg y) \equiv (y \Rightarrow x)\} \\
 & \neg(y \Rightarrow x) \\
 \equiv & \{(y \Rightarrow x) \text{ is } \neg y \vee x\} \\
 & \neg(\neg y \vee x) \\
 \equiv & \{\text{De Morgan, and the law of double negation}\} \\
 & y \wedge \neg x \\
 \equiv & \{\text{commutativity of } \wedge\} \\
 & \neg x \wedge y
 \end{aligned}$$

18. (a) Show that

$$\begin{aligned}
 (\forall i : q \wedge r : b) & \equiv (\forall i : q : r \Rightarrow b), \text{ and} \\
 (\exists i : q \wedge r : b) & \equiv (\exists i : q : r \wedge b).
 \end{aligned}$$

(b) Which of the following are valid?

$$\begin{aligned}
 (\forall x : (\exists y :: P(x, y))) & \equiv (\exists y : (\forall x :: P(x, y))). \\
 (\exists x : (\exists y :: P(x, y))) & \equiv (\exists y : (\exists x :: P(x, y))). \\
 (\forall x : (\forall y :: P(x, y))) & \equiv (\forall y : (\forall x :: P(x, y))).
 \end{aligned}$$

(c) Prove that all of the following expressions are equivalent.

$$\begin{aligned}
 & \neg(\exists x :: (\forall y :: P(x, y))), \\
 & (\forall x :: \neg(\forall y :: P(x, y))) \\
 & (\forall x :: (\exists y :: \neg P(x, y))).
 \end{aligned}$$

(d) Why do the following identities fail to hold even when p does not name i ?

$$\begin{aligned}
 p \wedge (\forall i : q : b) & \equiv (\forall i : q : p \wedge b) \\
 p \vee (\exists i : q : b) & \equiv (\exists i : q : p \vee b)
 \end{aligned}$$

Solution For the empty range,

$p \wedge (\forall i : q : b)$ is p and $(\forall i : q : p \wedge b)$ is *true*. Assign *false* to p to see that these two are different. Also, for the empty range,

$p \vee (\exists i : q : b)$ is p and $(\exists i : q : p \vee b)$ is *false*. Assign *true* to p to see that these two are different.

(e) Write the following statements formally.

- i. Every integer is bigger than some integer and smaller than some integer.

- ii. There is no integer that is bigger than all integers.
- iii. For all nonzero integers, there is a different integer having the same absolute value. (Use $|x|$ for the absolute value of x .)
- iv. No integer is both bigger and smaller than any integer.

Solution

- i. $(\forall x : x \text{ integer} :$
 $(\exists y : y \in \text{integer} : x > y)$
 $\wedge (\exists z : z \in \text{integer} : x < z)$
 $)$
- ii. $\neg(\exists x : x \in \text{integer} :$
 $(\forall y : y \in \text{integer} : x > y)$
 $)$
- iii. $(\forall x : x \in \text{integer} \wedge x \neq 0 :$
 $(\exists y : y \in \text{integer} \wedge x \neq y : |x| = |y|)$
 $)$
- iv. $\neg(\exists x : x \in \text{integer} :$
 $(\exists y : y \in \text{integer} : x > y \wedge x < y)$
 $)$

(f) Formally express the following facts about binary relation $*$ on set S .

- i. $*$ is reflexive,
- ii. $*$ is symmetric,
- iii. $*$ is transitive.

Solution

- i. $(\forall x : x \in S : x * x)$
 - ii. $(\forall x, y : x \in S, y \in S : x * y \Rightarrow y * x)$
 - iii. $(\forall x, y, z : x \in S, y \in S, z \in S : x * y \wedge y * z \Rightarrow x * z)$
- (g) Let $\leq$ be a partial order on set S . I defined *minimal* and *least* informally in Exercise 13 (page 76), where *minimal*(x, S) means that no element of S is smaller than x , and *least*(x, S) means that all elements of S are at least as large as x .
- Define *minimal*(x, S) and *least*(x, S) formally. Then, write the following statements formally.

- i. The least element of S is unique,
- ii. a least element of S is a minimal element of S , and
- iii. every subset of S , except the empty set, has a minimal element.

Solution

$minimal(x, S) :: x \in S \wedge \neg(\exists y : y \in S \wedge y \neq x : y \leq x)$

$least(x, S) :: x \in S \wedge (\forall y : y \in S : x \leq y).$

- i. $(\forall u, v : u \in S, v \in S : least(u, S) \wedge least(v, S) \Rightarrow u = v)$
- ii. $(\forall u : u \in S : least(u, S) \Rightarrow minimal(u, S))$
- iii. $(\forall T : \emptyset \subset T \subseteq S : (\exists u : u \in T : minimal(u, T)))$

19. Consider the following variation of the problem of pairing points with non-intersecting lines, Section 2.8.1.2 (page 60) with an additional requirement. Suppose each point is either *blue* or *red*, and there are equal number of blue and red points in a plane where no three points are on a line. Prove that it is possible to connect each blue point with a unique red point so that the connecting line segments are intersection-free.

Solution Figure 2.11(a) (page 60) offers two possible redirections: connect (a, c) and (b, d) (see Figure 2.11(b)) or (a, d) and (b, c) . One of these redirections connects blue points to red points, the other connects blue to blue and red to red. Either redirection decreases the combined lengths of the line segments. So, choose the one that connects points of different colors.

20. (Diagonalization) Show that the set of real numbers is not countably infinite.

Solution I show that even the set of real numbers between 0 and 1 is not countably infinite. Represent each number other than 1 as a fraction in binary and pad zeroes at its right end, if necessary, to form an infinite binary string. Represent 1 by the fraction that is an infinitely repeating sequence of all 1s, because $1 = (\sum_{i=0}^{\infty} 1/2^i)$. Thus, every number between 0 and 1 is uniquely represented by an infinite binary string, and each such string represents a distinct number between 0 and 1.

Let r_i be the bit representation of number i and r_{ij} be the bit at position j of r_i . Let b be an infinite binary string such that $b_j \neq r_{ij}$. Then, $b \neq r_i$ for any i because $b_i \neq r_{ii}$, that is, b and r_i differ at position i . Therefore, b does not appear in this list, that is, some number between 0 and 1 is not represented in the list, contradiction.

If each r_i is written horizontally and the list of r_i s enumerated vertically, then the list is a matrix of bits. Each row, an r_i , differs from b in its diagonal position. Hence, the name of the technique, diagonalization.

21. Show that the set of finite sequences of natural numbers is countably infinite. It then follows that the set of all finite subsets of natural numbers is countably infinite. The proof extends to finite sequences over any countably infinite set, not just natural numbers.

Hint Use the unique prime factorization theorem, Section 3.4.3 (page 104).

Solution It can be shown that the set of sequences of natural numbers of size k is countably infinite, and the union of these sets is again countably infinite. This proof requires some additional facts about countably infinite sets, so I give below a more elementary constructive proof.

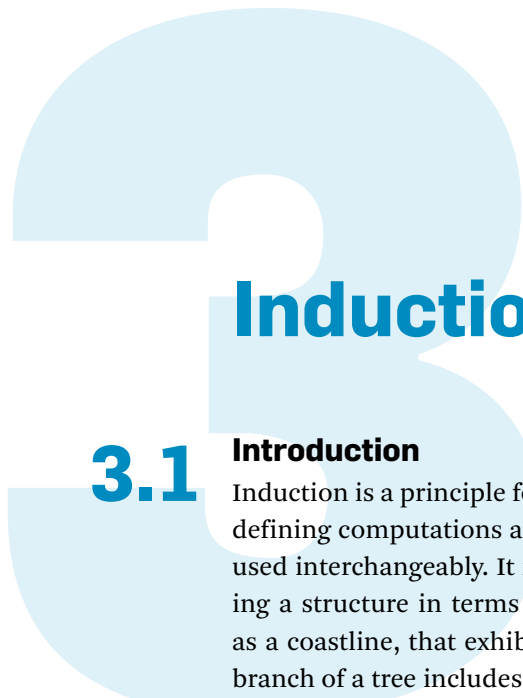
Let the sequence of natural numbers $\langle s_0, s_1, \dots, s_m \rangle$ correspond to the bag that has s_i occurrences of the i^{th} prime number (take the 0^{th} prime number to be 2). Note that the empty sequence corresponds to the empty bag. So, there is a 1–1 correspondence between finite sequences of natural numbers and bags of primes. Now use the unique prime factorization theorem (Section 3.4.3, page 104), which establishes a 1–1 correspondence between bags of primes and positive integers. Therefore, there is a 1–1 correspondence between finite sequences of natural numbers and positive integers, so the former set is countably infinite.

22. (Follow-up to Section 2.8.7, page 68) Prove for the least upper bound, $\uparrow$:
- (a) (Idempotence) $x \uparrow x = x$.
- (b) (Monotonicity) $a \leq x \wedge b \leq y \Rightarrow a \uparrow b \leq x \uparrow y$.

Proof of monotonicity Suppose $a \leq x \wedge b \leq y$. For any w ,

$$\begin{aligned}
 & x \uparrow y \leq w \\
 \equiv & \quad \{\text{Def } \uparrow\} \\
 & x \leq w \wedge y \leq w \\
 \Rightarrow & \quad \{\text{Premise: } a \leq x \wedge b \leq y, \text{ and transitivity of } \leq\} \\
 & a \leq w \wedge b \leq w \\
 \equiv & \quad \{\text{Def } \uparrow\} \\
 & a \uparrow b \leq w
 \end{aligned}$$

Using proposition 1, $a \uparrow b \leq x \uparrow y$.



Induction and Recursion

3.1 Introduction

Induction is a principle for constructing proofs, and recursion is a mechanism for defining computations and data structures. They are intimately related and often used interchangeably. It is easier to appreciate recursion, which is a way of defining a structure in terms of itself. The structure can be an everyday object, such as a coastline, that exhibits the same structure at various levels of granularity. A branch of a tree includes subbranches each of which may be regarded as a branch exhibiting similar properties. Several art works of M. C. Escher show a picture that depicts a smaller version of itself within it, and thus ad infinitum, or as far as the eye can discern. Many mathematical structures, including geometric curves, data structures in computer science, and computational procedures, can be described recursively. Recursive programming is so ubiquitous in computer science that I devote two chapters, Chapters 7 and 8, to it.

Induction is used to prove facts about recursively defined structures. Since recursion is so prevalent in computer science, induction is also ubiquitous in the proofs that are required in the developments of computer algorithms. The technique for sequential program proving, described in Chapter 4, is based on induction. Most of the advanced work in computer science requires a thorough understanding of induction and recursion.

The induction principle, in rough terms, says that a fact f can be proved for all instances of a structure by showing: (1) f holds for the smallest instances, and (2) f holds for an instance by assuming f holds for all its smaller instances. To see that f then holds for all instances, argue by contradiction. Suppose f does not hold for an instance; consider the smallest such instance J for which f does not hold. Clearly, J is not a smallest instance, because of (1). And, f holds for all instances smaller than J by our choice of J ; so, according to (2), it holds for J as well, a contradiction. Formal application of induction requires a clear definition of problem instance and its smaller instances. I develop these ideas and apply the induction principle in a variety of examples in this chapter.

Motivating the induction principle Let us try proving the following identity for all $n, n \geq 1$: $1 + 2 + \dots + n = \frac{n \times (n+1)}{2}$. Let us start by verifying the identity for some small values of n .

• $n = 1$: $1 = \frac{1 \times 2}{2}$.

• $n = 2$: $1 + 2 = 3 = \frac{2 \times 3}{2}$.

• $n = 3$: Let us try something different this time. Instead of computing the value of a subexpression, we substitute a value for it that we have computed previously.

$$\begin{aligned}
 & 1 + 2 + 3 \\
 = & \text{\{from the last proof, } 1 + 2 = \frac{2 \times 3}{2}\text{\}} \\
 & \frac{2 \times 3}{2} + 3 \\
 = & \text{\{factor } \frac{3}{2} \text{ from both terms}\}} \\
 & \frac{3}{2} \times (2 + 2) \\
 = & \text{\{arithmetic}\}} \\
 & \frac{3 \times 4}{2}
 \end{aligned}$$

• $n = 4$: Use the same strategy as for $n = 3$.

$$\begin{aligned}
 & 1 + 2 + 3 + 4 \\
 = & \text{\{from the last proof, } 1 + 2 + 3 = \frac{3 \times 4}{2}\text{\}} \\
 & \frac{3 \times 4}{2} + 4 \\
 = & \text{\{factor } \frac{4}{2} \text{ from both terms}\}} \\
 & \frac{4}{2} \times (3 + 2) \\
 = & \text{\{arithmetic}\}} \\
 & \frac{4 \times 5}{2}
 \end{aligned}$$

It is clear that we can continue constructing such proofs for $n = 5, 6 \dots$ ad infinitum, in each case relying on the last proof. So, we have proved the desired result, in principle, by imagining that any such proof can be written down.

But how can we convince a skeptic that “we can continue constructing such proofs for $n = 5, 6 \dots$ ad infinitum?” We can write down one typical proof and show that it fits the pattern of all such proofs. A proof for $n + 1$ will have the following structure:

• $n + 1$: To prove $1 + 2 + \dots + (n + 1) = \frac{(n+1) \times (n+2)}{2}$,

$$\begin{aligned}
 & 1 + 2 + \dots + (n + 1) \\
 = & \text{\{from the last proof, } 1 + 2 + \dots + n = \frac{n \times (n+1)}{2}\text{\}}
 \end{aligned}$$

$$\begin{aligned}
& \frac{n \times (n+1)}{2} + (n+1) \\
= & \{\text{factor } \frac{n+1}{2} \text{ from both terms}\} \\
& \frac{(n+1)}{2} \times (n+2) \\
= & \{\text{arithmetic}\} \\
& \frac{(n+1) \times (n+2)}{2}
\end{aligned}$$

The induction principle allows us to codify this process in a compact manner. We identify the smallest cases, in this example $n = 1$, and prove the desired result. Then write a proof, such as the one above for $n + 1$, that shows the typical proof structure for the other cases. In such a proof, replace “from the last proof” in a justification by “using the induction principle”.

The methodology in applying induction to prove a proposition is to first prove the base cases and then show how to construct a proof for the general case given proofs for all smaller cases. Analogously, the methodology in applying recursion is to first construct solutions for the smallest cases and then combine the solutions for the smaller cases to construct one for the general case.

3.1.1 Weak Induction Principle Over Natural Numbers

Let P_n be a proposition that includes n , a natural number, as a free variable. To prove P_n , for all $n, n \geq 0$, we prove:

$$(0) P_0, \quad (1) P_0 \Rightarrow P_1, \quad (2) P_1 \Rightarrow P_2, \quad (3) P_2 \Rightarrow P_3, \quad \dots$$

It should be clear that if all these (infinite number of) proofs can be successfully carried out, then we would have proved P_n , for all $n, n \geq 0$. Beyond (0), each proposition is of the same form: $P_i \Rightarrow P_{i+1}$. The following induction principle over natural numbers, often called *weak induction*, formalizes this argument. To prove that P_n holds for all $n, n \geq 0$, follow these steps.

- (Base step, or basis) Prove P_0 , and
- (Induction step) prove $P_i \Rightarrow P_{i+1}$, for all $i, i \geq 0$.

The proposition P_i is called the “inductive” or “induction hypothesis”.

Note on applying the base step The induction principle describes how to prove P_n , for all n , starting with $n = 0$. If P is to be proved for all n starting at some other value, say $n = 2$, the base step has to prove P_2 instead of P_0 . In general, the base step may be adjusted to a value that depends on the problem. For example, it can be shown by induction that $2^n < 3^{n-1}$, for all $n, n \geq 3$. The proof starts the base step with $n = 3$, that is, by showing $2^3 < 3^2$.

The shift in the base step can be justified. For the given example, define Q_n to be P_{n+3} , for all $n, n \geq 0$. We are then proving Q_n , for all $n, n \geq 0$, using the induction principle. In a typical proof we do not introduce a new proposition Q , instead proving P_n with an appropriate base value of n .

The base step may sometimes require proving several separate cases each of which may be considered a smallest instance of a problem (see Section 3.2.1.3 for an example).

3.1.2 Strong Induction Principle Over Natural Numbers

In order to establish P_n for all $n, n \geq 0$, the weak induction requires proving:

(0) P_0 , and (1) $P_i \Rightarrow P_{i+1}$ for all $i, i \geq 0$,

whereas the strong induction principle requires proving:

(0) P_0 , and (1) $(\forall i : 0 \leq i < j : P_i) \Rightarrow P_j$, for all $j, j > 0$.

That is, the weak induction assumes only P_i in proving P_{i+1} , whereas the strong induction assumes all preceding predicates, P_0 through P_{j-1} , in proving P_j . Strong induction is also called *complete induction*.

Weak and strong induction principles over natural numbers are equivalent. One of them is chosen to solve a particular problem depending on how easy it is to apply in that particular case, as we show in some of the examples in this section.

To see the equivalence of the two principles, any proof using weak induction can be converted to a proof using strong induction: from the proof of $P_j \Rightarrow P_{j+1}$ in the induction step in weak induction, conclude that $(\forall i : 0 \leq i \leq j : P_i) \Rightarrow P_{j+1}$, for all $j, j > 0$, by strengthening the antecedent. This is the required proof in strong induction. Conversely, a proof using strong induction can be converted to one using weak induction: define Q_0 as P_0 and Q_j as $(\forall i : 0 \leq i < j : P_i)$, for $j > 0$, that is, $P_0 \wedge P_1 \wedge P_2 \cdots P_{j-1}$. Strong induction proves Q_0 and $Q_j \Rightarrow Q_{j+1}$, for all $j, j \geq 0$, an application of weak induction with Q .

Rewriting the strong induction principle There is a neat way of rewriting the strong induction principle that omits explicit mention of the base step. The only proof step is the induction step, which is written as:

$(\forall i : 0 \leq i < j : P_i) \Rightarrow P_j$, for all $j, j \geq 0$.

That is, prove (0) $true \Rightarrow P_0$, (1) $P_0 \Rightarrow P_1$, (2) $(P_0 \wedge P_1) \Rightarrow P_2, \dots$.

The base step still needs to be proved somehow as demanded by (0), but this rewriting casts the proof requirement in the same form as the inductive proof. This formulation of induction plays an essential role in the more general well-founded induction principle, described in Section 3.5.

Henceforth, I use merely the term “induction” in place of “weak”, “strong” and “complete” induction.

3.2 Examples of Proof by Induction

3.2.1 Examples from Arithmetic

3.2.1.1 Sum of Cubes

Prove that $(+i : 0 \leq i \leq n : i)^2 = (+i : 0 \leq i \leq n : i^3)$, for all $n, n \geq 0$.

(Base) $n = 0$: Both sides have value 0.

(Induction) Assume that $(+i : 0 \leq i \leq n : i)^2 = (+i : 0 \leq i \leq n : i^3)$ for any $n, n \geq 0$. I show $(+i : 0 \leq i \leq n+1 : i)^2 = (+i : 0 \leq i \leq n+1 : i^3)$.

$$\begin{aligned}
 & (+i : 0 \leq i \leq n+1 : i)^2 \\
 = & \text{\{rewriting\}} \\
 & ((+i : 0 \leq i \leq n : i) + (n+1))^2 \\
 = & \text{\{arithmetic\}} \\
 & (+i : 0 \leq i \leq n : i)^2 + 2 \times (+i : 0 \leq i \leq n : i) \times (n+1) + (n+1)^2 \\
 = & \text{\{Induction applied to the first term\}} \\
 & (+i : 0 \leq i \leq n : i^3) + 2 \times (+i : 0 \leq i \leq n : i) \times (n+1) + (n+1)^2 \\
 = & \text{\{$(+i : 0 \leq i \leq n : i) = (n \times (n+1))/2$, from the earlier proof\}} \\
 & (+i : 0 \leq i \leq n : i^3) + n \times (n+1) \times (n+1) + (n+1)^2 \\
 = & \text{\{simplify the last two terms\}} \\
 & (+i : 0 \leq i \leq n : i^3) + (n+1)^3 \\
 = & \text{\{rewrite\}} \\
 & (+i : 0 \leq i \leq n+1 : i^3)
 \end{aligned}$$

3.2.1.2 A Fundamental Theorem of Number Theory

The following identity, known as Bézout’s identity, is fundamental in number theory. Given positive integers m and n there exist integers a and b such that

$$a \times m + b \times n = \gcd(m, n)$$

where $\gcd$ is the greatest common divisor. I use the following properties of $\gcd$ in the proof: (1) $\gcd(n, n) = n$, and (2) for $m > n$, $\gcd(m, n) = \gcd(m - n, n)$.

I prove this result by strong induction on $m+n$. The base case starts at 2 because m and n are at least 1.

- $m+n = 2$: Then $(m, n) = (1, 1)$ and $\gcd(m, n) = 1$. Use $(a, b) = (1, 0)$, which satisfies $1 \times 1 + 0 \times 1 = 1$.

- $m+n > 2$:

Case $m = n$: Then $\gcd(m, n) = m$. Use $(a, b) = (1, 0)$, which satisfies $1 \times m + 0 \times n = m$.

Case $m > n$: (the case $n > m$ is analogous). Consider the pair of positive integers $(m - n, n)$. Since $m - n + n = m < m + n$, by the strong induction hypothesis, there exist a' and b' such that $a' \times (m - n) + b' \times n = \gcd(m - n, n)$.

$$\begin{aligned}
 & a' \times (m - n) + b' \times n = \gcd((m - n), n) \\
 \Rightarrow & \{m > n \Rightarrow \gcd((m - n), n) = \gcd(m, n)\} \\
 & a' \times (m - n) + b' \times n = \gcd(m, n) \\
 \Rightarrow & \{\text{rewriting the lhs}\} \\
 & a' \times m + (b' - a') \times n = \gcd(m, n)
 \end{aligned}$$

Letting $(a, b) = (a', b' - a')$, we have established the result for (m, n) .

3.2.1.3 Fibonacci Sequence

The *Fibonacci* sequence¹, where F_i is the i^{th} number in the sequence, $i \geq 0$, is given by: $F_0 = 0$, $F_1 = 1$, and $F_{i+2} = F_i + F_{i+1}$, for all i , $i \geq 0$.

There are many fascinating facts about Fibonacci numbers. I prove two employing induction. See Exercise 5 (page 126) and Exercise 6 (page 127) for a few more results.

An elementary result For any n , $n \geq 0$, $(+i : 0 \leq i < n : F_i) < F_{n+1}$.

- Base case, $n = 0$: show $0 < F_1$, that is, $0 < 1$.
- Base case, $n = 1$: show $F_0 < F_2$, or $0 < 1$. Note that both base cases have to be proved separately. The case for $n = 1$ cannot be proved inductively from that for $n = 0$.
- Inductive case: For $n > 1$ show $(+i : 0 \leq i < n + 1 : F_i) < F_{n+2}$.

$$\begin{aligned}
 & (+i : 0 \leq i < n + 1 : F_i) \\
 = & \{\text{rewriting}\} \\
 & (+i : 0 \leq i < n : F_i) + F_n \\
 < & \{\text{induction hypothesis}\} \\
 & F_{n+1} + F_n \\
 = & \{\text{definition of Fibonacci sequence}\} \\
 & F_{n+2}
 \end{aligned}$$

1. This sequence is named after Fibonacci of Pisa, a mathematician of the Middle Ages. The sequence was devised earlier by Acharya Hemachandra in India in connection with the number of cadences of length n in music. See Exercise 4 (page 126).

A closed form for Fibonacci numbers There is a closed form description for Fibonacci numbers known as the Euler–Binet formula. Let ϕ and $\hat{\phi}$ be $\frac{1+\sqrt{5}}{2}$ and $\frac{1-\sqrt{5}}{2}$, respectively². It is easy to prove that $\phi^2 = 1 + \phi$ and $\hat{\phi}^2 = 1 + \hat{\phi}$. The Euler–Binet formula for all $n, n \geq 0$, is:

$$F_n = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}.$$

Finding the closed form, the inductive hypothesis, is hard, whereas the proof by induction, below, is relatively easy.

- Base case, $n = 0$: show $F_0 = \frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}}$, that is, $0 = \frac{1-1}{\sqrt{5}}$, which is straightforward.
- Base case, $n = 1$: show $F_1 = \frac{\phi^1 - \hat{\phi}^1}{\sqrt{5}}$, that is, $1 = \frac{\phi - \hat{\phi}}{\sqrt{5}}$, which follows by substituting the values for ϕ and $\hat{\phi}$.
- Inductive case, $n \geq 0$: show $F_{n+2} = \frac{\phi^{n+2} - \hat{\phi}^{n+2}}{\sqrt{5}}$.

$$\begin{aligned}
 & \frac{\phi^{n+2} - \hat{\phi}^{n+2}}{\sqrt{5}} \\
 = & \quad \{\text{rewriting}\} \\
 & \frac{\phi^2 \cdot \phi^n - \hat{\phi}^2 \cdot \hat{\phi}^n}{\sqrt{5}} \\
 = & \quad \{\phi^2 = 1 + \phi \text{ and } \hat{\phi}^2 = 1 + \hat{\phi}\} \\
 & \frac{(1 + \phi) \cdot \phi^n - (1 + \hat{\phi}) \cdot \hat{\phi}^n}{\sqrt{5}} \\
 = & \quad \{\text{rewriting}\} \\
 & \frac{(\phi^n - \hat{\phi}^n)}{\sqrt{5}} + \frac{(\phi^{n+1} - \hat{\phi}^{n+1})}{\sqrt{5}} \\
 = & \quad \{\text{apply induction on both terms}\} \\
 & F_n + F_{n+1} \\
 = & \quad \{\text{Definition of Fibonacci}\} \\
 & F_{n+2}
 \end{aligned}$$

3.2.1.4 Harmonic Numbers

Harmonic number, h_n , for any positive integer n , is defined by $h_n = 1/1 + 1/2 + 1/3 \cdots + 1/n$. Show that the harmonic sequence diverges, that is, there is no bound b such that all h_n are less than b .

2. The number ϕ is known as the *golden ratio*.

I show that $h_{2^k} > k/2$, for all $k, k \geq 0$. The base case, $k = 0$, asks for a proof of $h_1 > 0$, which follows from $h_1 = 1$. For the inductive case,

$$\begin{aligned}
 & h_{2^{k+1}} \\
 = & \{\text{definition of harmonic numbers}\} \\
 & h_{2^k} + (+i : 2^k + 1 \leq i \leq 2^{k+1} : 1/i) \\
 \geq & \{\text{each term in the sum is at least } 1/(2^{k+1}) \text{ and there are } 2^k \text{ terms}\} \\
 & h_{2^k} + 2^k \times 1/(2^{k+1}) \\
 = & \{\text{simplify the second term}\} \\
 & h_{2^k} + 1/2 \\
 > & \{\text{induction hypothesis: } h_{2^k} > k/2\} \\
 & k/2 + 1/2 \\
 = & \{\text{arithmetic}\} \\
 & (k+1)/2
 \end{aligned}$$

3.2.2 Examples About Games and Puzzles

3.2.2.1 Trimino Tiling

The following problem shows not just a proof by induction but also how an inductive proof yields a constructive solution procedure. The following intuitively appealing problem description, called “Tiling Elaine’s kitchen”, is due to David Gries. The Grieses have a square 8×8 kitchen, with a refrigerator on one square. David’s wife Elaine wants to tile the kitchen with L-shaped tiles, of course, not putting a tile under the refrigerator. (Who puts a refrigerator in the middle of a kitchen? The Grieses are different!)

Let us convert the problem by abstracting away from the kitchen and the refrigerator. Given is an 8×8 board, like a chess board, each square of which is called a *cell*. A *trimino* piece, an L-shaped tile, covers exactly three cells, two along the row and one along the column, or two along the column and one along the row; see Figure 3.1(a) for its shape and Figure 3.1(b) for the partial tiling of a 4×4 board. It is given that one cell is *closed* and the remaining ones are *open*. Show that it is

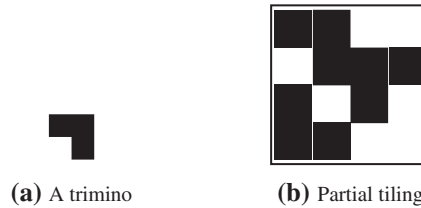


Figure 3.1 A trimino and partial tiling of a 4×4 board.

possible to tile the 8×8 board with triminos such that every open cell is covered by exactly one trimino and the closed cell is not covered.

The first step in solving this problem is to generalize it to a board of arbitrary size. For an $n \times n$ board, there are $n^2 - 1$ squares to be covered and a trimino covers three squares, so $n^2 - 1$ has to be a multiple of 3. It is true for $n = 8$ but not for arbitrary n . It is true for boards of size $2^n \times 2^n$; prove by induction that $2^n \times 2^n - 1$ is a multiple of 3, for all $n, n \geq 0$.

Then the general tiling problem is to show that a $2^n \times 2^n$ board with one closed cell can be covered by triminos, for all $n, n \geq 0$. Proof is by induction on n .

- (Base case) $n = 0$: the number of open cells is $2^0 \times 2^0 - 1 = 0$. Hence there is a trivial tiling that puts no tile at all.
- (Inductive case) $n + 1, n \geq 0$: Assume that any board of size $2^n \times 2^n$ with one closed cell can be tiled. Partition the given board of size $2^{n+1} \times 2^{n+1}$ into four equal sized subboards, each of size $2^n \times 2^n$. One of the subboards contains the closed cell. For each of the remaining three subboards pretend that its corner cell that is incident on the center of the board is closed (see Figure 3.2, page 91). Now, inductively, each subboard can be tiled because they have one closed cell each. Finally, tile all the pretend closed cells using one trimino (see Figure 3.2).

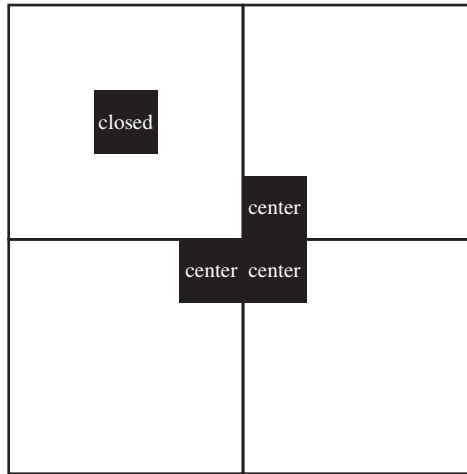


Figure 3.2 Tiling $2^n \times 2^n$ board with one closed and three pretend closed cells.

The proof tells us how to construct an actual tiling for any given instance of the problem. Such a solution is called a recursive solution, and this problem demonstrates the intimate connection between induction and recursion.

3.2.2.2 A Pebble Movement Game

Consider an odd number of cells arranged in a row where the center cell is empty, and each cell to the left of the empty cell has a black pebble and each cell to the right has a white pebble. There are two possible moves to rearrange the pebbles: (1) (shift) move a pebble to a neighboring cell if it is empty, and (2) (jump) jump a pebble over an adjacent pebble to land in an empty cell. It is required to show that all the black and white pebbles can be interchanged using only these moves.

To formalize the problem, consider a string over the alphabet $\{b, w, \circ\}$, where b denotes a black pebble, w a white pebble and $\circ$ the empty cell. Initially, the string is $b^n \circ w^n$, where b^n (w^n) is the n -fold repetition of b (w). The goal is to transform it to $w^n \circ b^n$ by suitable applications of shift and jump.

For strings P and Q , write $P \stackrel{s}{\sim} Q$ to denote that P can be transformed to Q by applying the shift rule once, $P \stackrel{j}{\sim} Q$ by applying the jump rule once, and $P \sim Q$ by applying both rules any number of times, including 0 times. Observe that shift and jump can be reversed, so all the given relations are symmetric. It is easy to show that $\sim$ is an equivalence relation over strings. Write the shift and jump rules formally; below p and q are single symbols, b or w :

$$p \circ \stackrel{s}{\sim} \circ p \quad \text{A pebble can move to the adjacent empty cell.} \quad (\text{R1})$$

$$pq \circ \stackrel{j}{\sim} \circ qp \quad \text{A pebble can jump over another to the empty cell.} \quad (\text{R2})$$

The goal is to show that $b^n \circ w^n \sim w^n \circ b^n$. I will actually prove a much stronger result. Let P be a string over any alphabet, not just the symbols $\{b, w, \circ\}$, that includes at least one $\circ$. Then P can be transformed to any of its permutations by multiple applications of (R1) and (R2). I prove this result in Lemma 3.2 for which I need Lemma 3.1, a generalization of (R1).

Lemma 3.1 $S \circ \sim \circ S$, for any string S .

Proof. Proof is by induction on the length of S .

- S is empty, show $\circ \sim \circ$: follows from the reflexivity of $\sim$.
- Inductive case, show $pS \circ \sim \circ pS$:

$$\begin{aligned} & pS \circ \\ \sim & \{ \text{Induction hypothesis: } S \circ \sim \circ S \} \\ & p \circ S \\ \sim & \{ \text{Rule R1: } p \circ \stackrel{s}{\sim} \circ p \} \\ & \circ pS \end{aligned}$$

■

Lemma 3.2 $P \sim Q$ where P and Q are permutations of each other.

Proof. I show that any two adjacent symbols pq in P can be transposed. This is sufficient to prove that the transformations can achieve any permutation (see Section 3.4.1, page 101). If at least one of p and q is $\circ$, then either they are both $\circ$ or a single application of (R1) transposes them. So, assume that neither p nor q is $\circ$.

Suppose pq precedes $\circ$ in P so that P is of the form $LpqC\circ R$. I show $LpqC\circ R \sim LqpC\circ R$. The proof for the case where $\circ$ precedes pq is analogous.

$$\begin{aligned}
 & LpqC\circ R \\
 \sim & \{\text{Lemma 3.1: } C\circ \sim \circ C, \text{ using } C \text{ for } S\} \\
 & Lpq\circ CR \\
 \sim & \{\text{Rule R2: } pq\circ \stackrel{j}{\sim} \circ qp\} \\
 & L\circ qpCR \\
 \sim & \{\text{Lemma 3.1: } \circ qpC \sim qpC\circ, \text{ using } qpC \text{ for } S\} \\
 & LqpC\circ R
 \end{aligned}$$

■

Lemma 3.2 does not explicitly use induction; induction is applied only in Lemma 3.1. I develop a complete proof of the original claim, $b^n \circ w^n \sim w^n \circ b^n$, in Exercise 16 (page 129).

3.3 Methodologies for Applying Induction

It is rare to find a situation, other than in a test, where you are supplied with an inductive hypothesis and asked to prove it. Proving a valid inductive hypothesis is typically easy; the more difficult aspects of induction lie in formulating the appropriate inductive hypothesis, or even knowing when and how to restate a problem to make it amenable to treatment by induction. There is no single recipe. I give a few hints in this section and show their applications on more advanced problems in Section 3.4.

3.3.1 Applying the Base Step

For induction over the natural numbers, it is usually preferable to apply the base step for $n = 0$, instead of $n = 1$, wherever possible. Consider the proof of $1 + 2 + \dots + n = \frac{n \times (n+1)}{2}$, for all positive n , which we saw in Section 3.1. We started with $n = 1$ as the base step and proved that $1 = \frac{1 \times 2}{2}$. The slightly more general proposition $(+i : 0 \leq i \leq n : i) = \frac{n \times (n+1)}{2}$ is also valid, and we could have proved the base step as $0 = 0$, taking $n = 0$. Then we would have avoided proving $1 = \frac{1 \times 2}{2}$, which is covered by the inductive step.

For this example, the additional overhead of starting the base at $n = 1$ is negligible. However, in some proofs the base step becomes elaborate if it has to mimic

the inductive step for a specific value. It is always preferable to start the induction at $n = 0$, or the smallest possible value, to avoid a superfluous proof that is covered by the inductive step.

3.3.2 Strengthening

I introduced strengthening (and weakening) of predicates in Section 2.5.3.2 (page 42). Predicate p *strengthens* predicate q , and q *weakens* p , if $p \Rightarrow q$. Perhaps the most important heuristic in applying induction is to suitably strengthen the inductive hypothesis. The inductive proof, $P_n \Rightarrow P_{n+1}$, may fail because P_n is too weak to establish P_{n+1} . Strengthening P to Q , that is, postulating Q such that $Q_n \Rightarrow P_n$, for all n , may allow the proof of $Q_n \Rightarrow Q_{n+1}$ as the inductive hypothesis. If we can conclude Q_n for all n , then, from $Q_n \Rightarrow P_n$, P_n holds for all n as well.

The Half-step puzzle: An example of strengthening This is a well-known puzzle that is easily solved using strengthening. Given are two points in a plane, x and y , that are unit distance apart. A man is initially at x . His goal is to reach y by taking a sequence of steps, but the length of each step he takes is half the length of his previous step, and his first step is at most $1/2$ unit in length. Show that it is impossible for him to reach y .

We may assume that the man takes the longest allowed step each time; so, his step lengths are $1/2, 1/4, \dots$. Then show that $(+i : 1 \leq i \leq n : 2^{-i}) \neq 1$, for all non-negative integers n .

A proof by induction may be attempted as follows.

- $n = 0$: Show $(+i : 1 \leq i \leq 0 : 2^{-i}) \neq 1$, which follows because the interval of summation is empty, therefore it yields 0.
- $n + 1$ for $n \geq 0$: Show $(+i : 1 \leq i \leq n + 1 : 2^{-i}) \neq 1$, given that $(+i : 1 \leq i \leq n : 2^{-i}) \neq 1$.

Now, $(+i : 1 \leq i \leq n + 1 : 2^{-i}) = (+i : 1 \leq i \leq n : 2^{-i}) + 2^{-(n+1)}$. By the induction hypothesis, the first term in the right side differs from 1. However, that is not sufficient to conclude that the sum differs from 1.

Solve the problem by strengthening the hypothesis to: $(+i : 1 \leq i \leq n : 2^{-i}) + 2^{-n} = 1$, for all non-negative integers n . The strengthened hypothesis implies the desired result, that $(+i : 1 \leq i \leq n : 2^{-i}) \neq 1$, because $2^{-n} \neq 0$ for all non-negative integers n . The proof of the strengthened hypothesis is given next.

- $n = 0$: Show $(+i : 1 \leq i \leq 0 : 2^{-i}) + 2^{-0} = 1$. That is, show $(+i : 1 \leq i \leq 0 : 2^{-i}) + 2^{-0} = 1$. The first term is 0 because the interval of summation is empty, and the second term is 1; hence the result.

- $n + 1$, for $n \geq 0$:

$$\begin{aligned}
 & (+i : 1 \leq i \leq n + 1 : 2^{-i}) + 2^{-(n+1)} \\
 = & \{ \text{Arithmetic: modify the interval of summation} \} \\
 & (+i : 1 \leq i \leq n :: 2^{-i}) + 2^{-(n+1)} + 2^{-(n+1)} \\
 = & \{ \text{Arithmetic} \} \\
 & (+i : 1 \leq i \leq n :: 2^{-i}) + 2^{-n} \\
 = & \{ \text{Induction hypothesis} \} \\
 & 1
 \end{aligned}$$

3.3.3 Generalization

One of the simplest examples of strengthening is to replace a constant in the inductive hypothesis by a variable, that is, instead of proving a result for a specific value of some variable, try proving something more general for all values of the variable from which the given instance follows. This heuristic is sometimes called *generalization*. As a trivial example, suppose you are asked to show that the sum of first 30 positive integers is $(30 \times 31)/2$. Then prove the more general result $(+i : 1 \leq i \leq n : i) = (n \times (n+1))/2$ (undoubtedly, you can also prove the given assertion by computing the values of both sides of the identity, but that is an impractical procedure if 30 is replaced by an astronomically large number). We used generalization in trimino tiling, Section 3.2.2.1, from 8×8 board to $2^n \times 2^n$ board, for any $n, n \geq 0$.

Another heuristic for generalization of predicate P_i , over variable i , is to construct predicate $Q_{i,j}$ with an additional variable, j , such that: (1) $Q_{i,N} \Rightarrow P_i$, for some fixed constant N , for all i , and (2) Q can be proved inductively. I have not yet shown the induction principle over two variables. For the moment, we can apply induction to prove Q in two steps, as an *inner* and an *outer* induction. The inner induction proves $Q_{(0,n)}$, for all values of n , using induction on n ; we may treat this as the base case of the outer induction. Next, prove for any value of n , $Q_{(i,n)} \Rightarrow Q_{(i+1,n)}$ for all i , as the inductive step of the outer induction. An example of such a proof appears in Section 3.4.4. This is not the only way to prove Q ; I show a general theory in Section 3.5.

3.3.4 Problem Reformulation, Specialization

A general problem-solving strategy is to reformulate a problem, to solve a related but simpler problem from which the solution to the original problem can be deduced. The strategy is not limited to just induction, but many inductive proofs can benefit by first proving something simpler. The problem may be made simpler in a variety of ways, by abstracting from the original description, by adopting

appropriate notations, and sometimes by ignoring a few special cases. Solving suitably chosen special cases may lend insight to the solution of the general problem. This is particularly relevant for induction and recursion. I encourage problem-solvers to attempt enumerating the solutions for the base case and a few other small cases to observe the pattern of solutions and then generalize from it. I give examples of problem reformulation in Sections 3.4.2.2 (page 104), 3.4.3 and 3.4.5 that lead to significantly simpler proofs. Here I illustrate the idea with a simple example, the proof of the binomial theorem with non-negative integer exponent.

The binomial theorem for natural number n and real numbers x and y is:

$$(x + y)^n = (+i : 0 \leq i \leq n : \binom{n}{i} \cdot x^{n-i} \cdot y^i),$$

where $\binom{n}{i}$ is a binomial coefficient. The theorem can be proved using induction on n and certain identities over the binomial coefficients. First, I show a seemingly simpler result:

$$(1 + z)^n = (+i : 0 \leq i \leq n : \binom{n}{i} \cdot z^i).$$

Exercise 10 (page 128) asks you to prove this fact using induction. I prove the original theorem using this result. Multiply both sides of the simple result by x^n and also replace z by y/x (for the moment, assume that $x \neq 0$) to get:

$$\begin{aligned} x^n \cdot (1 + y/x)^n &= (+i : 0 \leq i \leq n : \binom{n}{i} \cdot x^n \cdot (y/x)^i), \text{ that is,} \\ (x + y)^n &= (+i : 0 \leq i \leq n : \binom{n}{i} \cdot x^{n-i} \cdot y^i), \text{ as required.} \end{aligned}$$

The remaining task is to prove the theorem for $x = 0$, that is, show

$y^n = (+i : 0 \leq i \leq n : \binom{n}{i} \cdot 0^{n-i} \cdot y^i)$, which is as follows. I will adopt the convention that $0^0 = 1$.

$$\begin{aligned} & (+i : 0 \leq i \leq n : \binom{n}{i} \cdot 0^{n-i} \cdot y^i) \\ = & \{ \text{rewrite the quantified expression} \} \\ & (+i : 0 \leq i < n : \binom{n}{i} \cdot 0^{n-i} \cdot y^i) + \binom{n}{n} \cdot 0^0 \cdot y^n \\ = & \{ \text{For positive } n, 0^{n-i} = 0 \text{ for } 0 \leq i < n. \\ & \text{For } n = 0, \text{ the quantified expression is} \\ & \text{a sum over an empty interval, so it is } 0 \} \\ & \binom{n}{n} \cdot 0^0 \cdot y^n \\ = & \{ \binom{n}{n} = 1; \text{ and } 0^0 = 1, \text{ by convention} \} \\ & y^n \end{aligned}$$

In summary, the reformulation resulted in a simpler problem that could be proved using induction. The original theorem is then proved, ignoring the corner case of $x = 0$, and the corner case is handled separately.

3.3.5 Proof by Contradiction versus Proof by Induction

A proof by contradiction can often be replaced by a proof by induction. In proving that P holds for all natural numbers, a typical proof by contradiction is: let n be the smallest natural number that violates P . The proof then displays i , $i < n$, such that P_i is violated, contradicting that n is the smallest natural number that violates P .

Formally, the proof by contradiction is $\neg P_n \Rightarrow (\exists i : 0 \leq i < n : \neg P_i)$, for all n , $n \geq 0$. Taking the contrapositive, this predicate is equivalent to: $(\forall i : 0 \leq i < n : P_i) \Rightarrow P_n$, for all n , $n \geq 0$. This is exactly the rewriting of the strong induction principle described in Section 3.1.2. Thus, every proof by contradiction in this form can be converted to a proof by induction (note that a strong induction proof is also a weak induction proof, as explained in Section 3.1.2).

Not all proofs by contradiction have a counterpart inductive proof. For example, Cantor's diagonalization argument (Exercise 20 of Chapter 2, page 81), is a proof by contradiction, but there is no corresponding proof by induction. If the proof by contradiction depends on some *well-founded ordering* over the set (see Section 3.5, page 115, for a discussion of well-founded ordering), as in “ n is the *smallest* natural number that violates P ”, it can often be replaced by an inductive proof. It is more instructive to construct a proof by induction rather than contradiction whenever possible.

3.3.6 Misapplication of Induction

Induction is often misapplied by novices and sometimes by the experts. I identify two root causes of misapplication, in the base and in the inductive step.

Pay attention to the base case Sometimes the base step is completely omitted, but this is rare in published proofs. The more likely scenario is that the inductive step assumes something about the base step that has not been proved. I show two such proofs of absurd results. The mistake in such a proof can be detected by instantiating the inductive proof for a few small cases.

Consider the following “proof” of $x^n = 1$, for any positive real x and any natural n . Induction is on n .

- $n = 0$: $x^0 = 1$ follows from arithmetic.
- $n + 1$, where $n \geq 0$:

$$\begin{aligned}
 & x^{n+1} \\
 = & \quad \{\text{Arithmetic}\} \\
 & \frac{x^n \times x^n}{x^{n-1}} \\
 = & \quad \{\text{Inductive hypothesis: all the terms are 1}\}
 \end{aligned}$$

$$\begin{aligned}
 & (1 \times 1)/1 \\
 = & \frac{\{\text{Arithmetic}\}}{1}
 \end{aligned}$$

The proof is incorrect in replacing x^{n-1} by 1. This is because for the case $n = 0$, x^{n-1} has not been shown to be equal to 1. The mistake could have been detected by instantiating the inductive step with $n = 0$, which claims $\frac{x^0 \times x^0}{x^{0-1}} = (1 \times 1)/1$. Another way to tell such an error is that it causes a “type violation”, that $n - 1$ is not always a natural number given that $n \geq 0$.

The next puzzle is given in an informal style (that may be the cause of the error). I show that all children in a town are of the same height. The proof is by induction on the number of children. Start the induction with one child, who is of the same height as herself. Next consider a set of $n + 1$ children where $n \geq 1$. Consider two different subsets of n children of this set. By the inductive hypothesis, children in each subset are of the same height, and the two subsets have a common child. So, all children in the set of $n + 1$ children are of the same height.

The proof considers two different subsets of children, and assumes that the two subsets share a child c . Then each child in each subset has the same height as c , inductively; therefore all $n + 1$ children have the same height. The proof works for $n + 1$ children, $n \geq 2$. But it does not hold for $n + 1$ where $n = 1$; instantiate the inductive proof for this case to detect the fault.

Interestingly, if we include the case for $n = 2$ as a hypothesis, that every pair of children are of the same height, then all children are of the same height by the inductive proof given above.

Ordering of subproblems The induction principle, in rough terms, is as follows. To prove that every instance of a problem satisfies predicate P , prove that P holds for the smallest instances, and if P holds for all subproblems of any problem instance J , it also holds for J (which is in the spirit of the strong induction principle). A formalization of this idea gives the most general principle of induction, as explained in Section 3.5 (page 115). The definition of a subproblem is rigorously defined in that section by introducing an ordering over the problems. Informal proofs often avoid the rigor and substitute a common-sense notion of subproblem. I show an example below.

I “prove” by induction that the shortest path between any two points in the Euclidean plane is a straight line, a result that is usually proved using calculus. Let x and y be two points in a plane, not necessarily distinct, and let d be the minimum distance between x and y along any path connecting them. The proof is by induction on d .

For $d = 0$, we have $x = y$, so any path other than a straight line has greater length than d .

For $d > 0$, let z be a point on the shortest path between x and y , $z \neq x$, $z \neq y$; see Figure 3.3 where the purported shortest path is shown as a finely dotted curve. Each of the distances xz and yz along the given shortest path is strictly smaller than d . Applying induction, the shortest paths between x and z and between z and y are straight lines, shown as thick dashed lines. Let $|pq|$ denote the straight-line distance between any two points p and q . Then the shortest path length between x and y , which passes through z , is $|xz| + |zy|$. From the triangle inequality, $|xy| \leq |xz| + |zy|$, and $|xy|$ is strictly smaller than $|xz| + |zy|$ unless z is on the straight line xy . Thus, the straight line xy is the shortest path, shown as a heavy solid line.

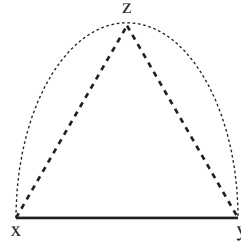


Figure 3.3 A “proof” that straight line is the shortest path.

This proof is wrong even though the result is correct. The proof does not follow the induction principle of defining a predicate over natural numbers. You cannot apply induction on real numbers as we have done with d in this example.

In my experience, many people are undeterred even if they are told that this proof by induction is incorrect. You cannot refute their argument by displaying a counterexample, because the result in this example is correct. A useful counterargument is to show that their proof technique can be used to prove something that is known to be wrong. For this case, I recommend applying the incorrect argument to the half-step puzzle given in Section 3.3.2 and prove that the man can indeed reach his goal y , which has been shown to be impossible. The incorrect argument applies induction on the remaining distance from the man to y . Let the current position of the man be z ; if $z = y$, the man has reached his goal, otherwise he takes the longest permissible step toward y after which the distance to y is shorter, and, by induction, he can reach y eventually.

Don’t overuse induction Any argument that ends with “and repeat these steps until ...” or something similar is a candidate for an inductive proof. However, not everything needs to be formal. If I can always take a step of unit length, then I

can eventually reach any destination at a finite distance. And if a program processes the items in a file sequentially, it will eventually process all the items. We don't need inductive proofs in such cases, same as we don't need any proof when common-sense reasoning is overwhelmingly convincing.

There are different inductive proofs for the same problem The elementary examples we have seen so far may lead us to believe that there is just one way to apply induction for a given problem. That is definitely not true. In fact, exploring alternatives is one of the best way to truly understand a problem.

I proved Bézout's identity in Section 3.2.1.2 (page 87) where we had to apply induction on two variables, m and n . Since I had not yet developed the theory for two-variable induction, I based the inductive proof on $m + n$. Now, summation of m and n has nothing at all to do with the problem; all we need is to prove the result for increasing values of m and n . We could have used $m \times n$, a far worse alternative. The proper way is to apply lexicographic ordering over pairs of values, a topic I discuss in Section 3.5.1.2 (page 117).

An identity about Fibonacci numbers (Exercise 5d, page 127) can be proved directly or more easily by using matrix algebra (see Exercise 6, page 127). Inductive proofs about trees are usually based on the number of nodes, number of edges or the maximum path length, and in some cases any one of them can be used. For matrix-based problems, similarly, many different measures are available for inductive proofs. The lesson is, don't be too happy with your first success, explore.

Proofs over infinite structures I emphasize that the induction principle, as given, is applicable only over the set of natural numbers. So, you can only prove P_n for all natural numbers n , for some predicate P . The principle does not extend to proving P over an infinite structure. To see this, consider proving that every finite binary sequence has a finite number of 0s. This is easily proved by using induction on the length of the sequence. However, we can *not* conclude that every infinite binary sequence has a finite number of 0s, which, of course, is wrong.

Next, consider a less trivial problem. An *infinite tree* has an infinite number of nodes. Suppose that in a given infinite tree if there is a path of length n from the root there is also a path of length $n + 1$, for any natural number n . We can then show, using induction, that there are paths of all possible lengths (note that the root is at distance 0). But we can not assert that there is an infinite path. To see a counterexample to this assertion, number the root node 0, let the root have an infinite number of children numbered with successive positive integers, and node i , $i > 0$, is the root of a chain of length $i - 1$. Thus, the root has a path of

length i that passes through node i , for every positive i . However, there is no infinite path.

There are ways of applying induction to prove facts about infinite structures. I do not go deeply into this topic in this book, but I consider a problem over infinite trees in Section 3.4.8 and prove the famous König's lemma that asserts the existence of infinite paths in certain infinite trees.

3.4 Advanced Examples

The examples of induction we have seen so far have been quite simple. In most cases the inductive hypothesis was given, and the goal was to complete the steps of the proof. In other cases the inductive hypothesis was easy to formulate. This section contains a few examples in which the inductive hypothesis is not immediately available or obvious. Their solutions illustrate some of the techniques and methodologies I have outlined in Section 3.3.

3.4.1 Permutation Using Transposition

I show that a sequence can be rearranged to any desired permutation by suitably transposing (i.e., exchanging) adjacent pairs of items. Specifically, let P be a sequence of items, not necessarily distinct, and Q a permutation of P , as in $P = \langle 3\ 2\ 7\ 1 \rangle$ and $Q = \langle 1\ 2\ 3\ 7 \rangle$. We can apply the following sequence of transpositions to convert P to Q : $\langle (3, 2)\ (7, 1)\ (3, 1)\ (2, 1) \rangle$.

The challenge in this problem is to find a variable on which we can apply induction. The description of the problem does not readily suggest such a variable. To simplify the problem, assume that we are dealing with sequences of distinct integers. A pair of items (x, y) in a sequence is an *inversion* if they are out of order, that is, x precedes y in the sequence and $x > y$. So, $(3, 2)$, $(3, 1)$ and $(7, 1)$ are inversions in P , given above. I show that if adjacent pairs of items that form inversions are repeatedly transposed, then the sequence will be sorted in increasing order eventually. This procedure can be slightly modified to solve the original problem, to convert a sequence to any permutation through adjacent transpositions.

Let us formulate the inductive hypothesis: any sequence with n inversions can be sorted through adjacent transpositions. The base case is easy; if $n = 0$ there is no inversion, so the sequence is already sorted. For $n > 0$, suppose (x, y) is an inversion, and x and y are adjacent. What is the effect on n of transposing x and y ? Every inversion that does not include both x and y remains an inversion, because x and y are adjacent. And the inversion (x, y) is eliminated through the transposition, so n decreases. Inductively, the resulting permutation can be sorted through adjacent transpositions.

The original problem of converting an arbitrary sequence P to a permutation Q can be recast as a sorting problem over integers. Assign increasing natural numbers as indices to the items of Q , so Q is sorted according to the indices. Each item of P receives the index of an identical item in Q ensuring that all indices are distinct, which is possible because P is a permutation of Q . Then convert P to Q by applying the sorting procedure.

The transpositions need not be over adjacent items; transposition of the items in any inversion also decreases the number of inversions. However, this result is proved more easily using lexicographic ordering (see Section 3.5.1.2, page 117).

This example shows an elementary form of problem reformulation, of casting the original problem as a sorting problem. More significantly, it shows that the induction variable may not be apparent from the problem description.

3.4.2 Arithmetic Mean Is At Least the Geometric Mean

The arithmetic mean of a finite non-empty bag of non-negative real numbers is greater than or equal to its geometric mean. This classical result, sometimes called the *am-gm inequality*, has been proved for many centuries. I show two proofs, the first one due to the famous mathematicians Hardy, Littlewood and Pólya that uses an interesting use of induction. The second proof uses problem reformulation (described in Section 3.3.4) to reduce the proof to a very simple one using standard induction. An ingenious proof by Dijkstra in EWD1231 [Dijkstra 1996a], based on the ideas of program proving, is nearly as simple as the second proof given here (see Section 5.3, page 207).

For a bag that contains a 0, the geometric mean is 0, whereas the arithmetic mean is non-negative, so the theorem holds. Henceforth, assume that the elements of the bags are positive real numbers.

3.4.2.1 A Proof Due to Hardy, Littlewood and Pólya

First, prove the result for all bags whose sizes are powers of 2. Then show that if the result holds for all bags of size n , $n > 2$, it holds for all bags of size $n - 1$ as well. This is an inductive step in the reverse direction, and it establishes that the result holds for all bags of size lower than n provided it holds for n . Since the result holds for all bags of size 2^n , $n \geq 0$, it holds for any bag of size k because $k \leq 2^n$ for some n .

First, I prove the result for bags of sizes 1 and 2 as base cases, and then for bags of size 2^n , $n > 1$, by induction.

- Bag size = 2^0 : Both the arithmetic and the geometric mean are equal in value to the bag element.

- Bag size = 2^1 : Suppose the bag elements are x and y , so the arithmetic mean is $(x + y)/2$ and the geometric mean is $\sqrt{x \cdot y}$.

$$\begin{aligned}
 & ((x + y)/2)^2 \\
 = & \quad \{\text{algebra}\} \\
 & ((x - y)^2 + 4 \cdot x \cdot y)/4 \\
 \geq & \quad \{(x - y)^2 \geq 0\} \\
 & (4 \cdot x \cdot y)/4 \\
 = & \quad \{\text{simplify}\} \\
 & x \cdot y
 \end{aligned}$$

Taking square roots of the top and bottom expressions, since x and y are positive reals, $(x + y)/2 \geq \sqrt{x \cdot y}$.

- Bag size = 2^{n+1} , $n > 0$:

Divide the bag into two bags of sizes 2^n each whose arithmetic means are a and a' and geometric means are g and g' , respectively. So, for the whole bag the arithmetic and geometric means are $(a + a')/2$ and $\sqrt{g \cdot g'}$, respectively. I show $(a + a')/2 \geq \sqrt{g \cdot g'}$.

$$\begin{aligned}
 & (a + a')/2 \\
 \geq & \quad \{\text{inductively, } a \geq g \text{ and } a' \geq g'\} \\
 & (g + g')/2 \\
 \geq & \quad \{\text{consider bag } \{g, g'\} \text{ of two elements;} \\
 & \text{using } g, g' \text{ for } x, y \text{ in the last proof}\} \\
 & \sqrt{g \cdot g'}
 \end{aligned}$$

■

Next, I show that if the result holds for all bags of size n , $n > 2$, it holds for all bags of size $n - 1$ as well. Consider bag X of size $n - 1$ whose sum of elements is A , product G , arithmetic mean a and geometric mean g . Then, $a = (1/(n - 1))A$ and $g = G^{1/(n-1)}$. Define bag Y by $Y = X \cup \{a\}$. The sum and product of Y are $A + a$ and Ga , respectively. Then the arithmetic mean of Y is $(1/n)(A + a) = (1/n)(A + (1/(n - 1))A) = (1/(n - 1))A = a$. And the geometric mean of Y is $(Ga)^{1/n}$. The size of Y is n . From the inductive hypothesis $a \geq (Ga)^{1/n}$. Our goal is to show the result for X , that $a \geq g$.

$$\begin{aligned}
 & a \geq (Ga)^{1/n} \\
 \Rightarrow & \quad \{\text{algebra}\} \\
 & a^n \geq Ga \\
 \Rightarrow & \quad \{\text{algebra}\} \\
 & a^{n-1} \geq G \\
 \Rightarrow & \quad \{\text{algebra}\}
 \end{aligned}$$

$$\begin{aligned}
& a \geq G^{(1/(n-1))} \\
\Rightarrow & \{g = G^{(1/(n-1))}\} \\
& a \geq g
\end{aligned}$$

3.4.2.2 A Simple Proof Using Reformulation

The following proof is a slightly simpler version of the one in [Beckenbach and Bellman \[1961, section 11\]](#).

The essential idea is “scaling”, multiplying all elements of a bag by a positive real number. Scaling by r converts the arithmetic mean a and geometric mean g to $r \cdot a$ and $r \cdot g$, respectively, and $a \geq g \equiv r \cdot a \geq r \cdot g$. Choose r to be the reciprocal of the product of the elements of the bag, so the product after scaling is 1 and $g = 1$. It is then sufficient to show that the arithmetic mean is at least 1. Equivalently, prove that for any non-empty bag B of n positive real numbers whose product is 1, the sum of its elements, $\text{sum}(B)$, is at least n . Proof is by induction on n .

For $n = 1$, the arithmetic and geometric mean are identical, so the result holds. For the inductive step let B have $n + 1$ elements, $n \geq 1$. Choose distinct elements x and y from B where x is a smallest and y a largest element. Since the product of the elements is 1, $x \leq 1$, and, similarly, $y \geq 1$. Let B' be the bag obtained by replacing x and y in B by their product, that is, $B' = B - \{x, y\} \cup \{x \times y\}$. Note that the size of B' is n and the product of the elements of B' remains equal to 1; so, inductively $\text{sum}(B') \geq n$. Our goal is to prove that $\text{sum}(B) \geq n + 1$.

$$\begin{aligned}
& \text{sum}(B) \\
= & \{\text{Definition of } B'\} \\
& \text{sum}(B') + x + y - x \times y \\
\geq & \{x \leq 1, y \geq 1 \Rightarrow (1 - x)(y - 1) \geq 0, \text{ or } x + y - x \times y \geq 1\} \\
& \text{sum}(B') + 1 \\
\geq & \{\text{sum}(B') \geq n, \text{ from the induction hypothesis}\} \\
& n + 1
\end{aligned}$$

Observe that the proof does not require x and y to have different values; i.e. there is no separate case analysis for all elements of B being equal to 1.

3.4.3 Unique Prime Factorization

The unique prime factorization theorem, sometimes called the fundamental theorem of arithmetic, says: every positive integer can be written as a product of primes in a unique way. We may be tempted to formalize the statement of the theorem in this manner: for every positive integer n there exist non-negative exponents $e_1, e_2 \dots e_m$, for some positive integer m , such that $n = \prod_{1 \leq i \leq m} p_i^{e_i}$, where p_i is the i^{th} prime number; further, the exponents e_i are unique. This description has neither

the virtue of simplicity of the informal statement nor the properties we need for use in mathematical manipulations. The informal statement is easy to understand but it gives little guidance about how to use it in a formula.

Observe that in a factorization of a number a specific prime may occur more than once, so the factorization yields a *bag* of primes. The theorem claims that this bag is unique. The easy part of the theorem is that every number has a factorization into primes, that is, for every integer n there is a bag of primes R such that the product of its elements, ΠR , is n . I leave this proof to the reader as an exercise in applying strong induction. I prove the uniqueness part in this section.

The uniqueness requirement is surprisingly easy to formalize: if bags R and S are factorizations of the same number, then $R = S$. Since $R = S$ implies $\Pi R = \Pi S$, we can dispense with n altogether, and state the uniqueness part of the theorem: for bags of primes R and S , $R = S$ if and only if $\Pi R = \Pi S$. The proof shows not just the usefulness of induction but also the importance of choosing appropriate notations.

Notation In this proof lowercase letters like p and q denote primes and uppercase ones like S and T denote finite bags of primes. Write $p \mid x$ for “ p divides x ”. The product of the elements of an empty bag, $\Pi \{\}$, is 1, the unit element of the product operator. Therefore, the unique bag corresponding to 1 is $\{\}$.

Lemma 3.3 $p \mid \Pi S \equiv p \in S$.

Proof. It is easy to see the proof in one direction: $p \in S \Rightarrow p \mid \Pi S$. I prove $p \mid \Pi S \Rightarrow p \in S$, that is, every prime divisor of a positive integer is in every factorization bag of it, by induction on the size of S .

- $S = \{\}$: Then, $p \mid \Pi S$ is *false* for every prime p , and the hypothesis is true vacuously.
- $S = T \cup \{q\}$, for some bag of primes T and prime q : If $p = q$, then $p \in S$, so the result holds trivially. For $p \neq q$ employ Bézout’s identity, given in Section 3.2.1.2: for any pair of positive integers m and n there exist integers a and b such that $a \cdot m + b \cdot n = \gcd(m, n)$. Using p and q for m and n , respectively, and noting that $\gcd(p, q) = 1$ for distinct primes, we have $a \cdot p + b \cdot q = 1$ for some a and b .

$$\begin{aligned}
 & p \mid \Pi S \\
 \Rightarrow & \{p \mid a \cdot p \cdot \Pi T \text{ and } p \mid \Pi S. \text{ So, } p \mid (a \cdot p \cdot \Pi T + b \cdot \Pi S)\} \\
 & p \mid (a \cdot p \cdot \Pi T + b \cdot \Pi S) \\
 \Rightarrow & \{S = T \cup \{q\}. \text{ So, } \Pi S = q \cdot \Pi T\}
 \end{aligned}$$

$$\begin{aligned}
& p \mid \Pi T(a \cdot p + b \cdot q) \\
\Rightarrow & \{a \cdot p + b \cdot q = 1\} \\
& p \mid \Pi T \\
\Rightarrow & \{\text{inductive hypothesis}\} \\
& p \in T \\
\Rightarrow & \{T \subseteq S\} \\
& p \in S
\end{aligned}$$

Theorem 3.1 Unique prime factorization

$$(R = S) \equiv (\Pi R = \Pi S).$$

Proof. We can argue inductively, based on Lemma 3.3, that the bag corresponding to a number x is unique: if $p \mid x$, then p is in the bag and x/p has a unique bag, by induction; and if p does not divide x , then p is not in the bag. So, the bag corresponding to x is unique. I show a formal proof next.

Obviously, $(R = S) \Rightarrow (\Pi R = \Pi S)$. I prove that if $(\Pi R = \Pi S)$, then R and S are equal as bags. It is easy to show that for any p , $(p \in R) \equiv (p \in S)$. This proves that R and S have the same *set* of elements, as in $R = \{2, 2, 3\}$ and $S = \{2, 3, 3\}$, not the same *bag* of elements.³ The following proof uses induction on n , the size of R .

- $n = 0$: Then R is the empty bag, so $\Pi R = 1 = \Pi S$. Then S is the empty bag.
- $n > 0$: R , being non-empty, has an element p .

$$\begin{aligned}
& p \in R \\
\equiv & \{\text{from Lemma 3.3}\} \\
& p \mid \Pi R \\
\equiv & \{\Pi R = \Pi S\} \\
& p \mid \Pi S \\
\equiv & \{\text{from Lemma 3.3}\} \\
& p \in S
\end{aligned}$$

Let $R' = R - \{p\}$ and $S' = S - \{p\}$. Inductively, $R' = S'$ as bags. So, $R = S$ as bags because $R = R' \cup \{p\}$ and $S = S' \cup \{p\}$. ■

3.4.4 Termination of a Game

The following puzzle combines multiple applications of induction. Given is a set of integer-valued variables x_i , where $0 \leq i < 2^n$ and $n \geq 1$. The right neighbor of variable x_i is $x_{i+1 \pmod{2^n}}$, that is, the variables are arranged circularly. In each step of the game each variable's value is replaced by the absolute value of the difference between it and its right neighbor, that is, x_i is assigned $|x_i - x_{i+1 \pmod{2^n}}|$,

3. This mistake in my original proof was spotted by Rutger Dijkstra.

simultaneously for all i , $0 \leq i < 2^n$. The game terminates when all variable values are equal to 0 because then there is no further change in the values. Show that starting with arbitrary initial values the game always terminates. For the proof, I let the game run forever even though there is no change in the values after they all become 0.

To simplify notation, use $+$ in the subscripts for addition modulo 2^n , so that $x_{i+1 \pmod{2^n}}$ is written as x_{i+1} .

Problem reformulation Observe that after the first step all values are non-negative. So, reformulate the problem by assuming that the starting values are non-negative. Then the maximum value in the ring never increases after a step because $|x_i - x_{i+1}| \leq \max(x_i, x_{i+1})$. Let the binary expansion of the initial maximum value have m bits. I prove the result by applying induction on m . The base case, $m = 1$, also needs an induction on n . In fact, the base case has a longer proof than the inductive case.

Base case, $m = 1$: With $m = 1$ the initial values are either 0 or 1, and according to the rules of the game they remain 0 or 1 after a step. The effect of a step is to set x_i to $x_i \oplus x_{i+1}$ where $\oplus$ is addition modulo 2. Note that $\oplus$ is commutative and associative, and $x \oplus x = 0$, for any x .

Let k be any power of 2, and x_i^k the value of x_i after k steps, for any i ; so, x_i^0 is the initial value of x_i . I show: $x_i^k = x_i^0 \oplus x_{i+k}^0$, for all i and k , by induction on k .

- $k = 1$:

Show $x_i^1 = x_i^0 \oplus x_{i+1}^0$. This is exactly the rule of the game.

- $2 \cdot k$, where $k \geq 1$: Show $x_i^{2 \cdot k} = x_i^0 \oplus x_{i+2 \cdot k}^0$.

Observe that $x_i^{2 \cdot k}$ is the result of applying k steps starting with the initial values of x_j^k for all j , so, inductively, $x_i^{2 \cdot k} = x_i^k \oplus x_{i+k}^k$.

$$\begin{aligned}
 & x_i^{2 \cdot k} \\
 = & \{ \text{from above observation} \} \\
 & x_i^k \oplus x_{i+k}^k \\
 = & \{ \text{induction hypothesis on both terms} \} \\
 & x_i^0 \oplus x_{i+k}^0 \oplus x_{i+k}^0 \oplus x_{i+k+k}^0 \\
 = & \{ x_{i+k}^0 \oplus x_{i+k}^0 = 0 \} \\
 & x_i^0 \oplus x_{i+2 \cdot k}^0
 \end{aligned}$$

It follows from the proof that $x_i^{2^n} = x_i^0 \oplus x_{i+2^n}^0$. Since x_{i+2^n} is x_i , $x_i^0 \oplus x_{i+2^n}^0 = 0$. So, the game terminates starting with initial values of 0 and 1.

Inductive case, $m > 1$: For $m > 1$ the game has the same effect on the last bit of every value as in $m = 1$. Therefore, eventually (after 2^n steps) all the last bits are 0. After that point the last bits remain 0, that is, all numbers remain even, because $|x_i - x_{i+1}|$ is even. We may imagine that the game is then played with values that

have only their leading $m - 1$ bits, because the last bits remain 0. Inductively, this game terminates eventually. Therefore, the game with m bit numbers terminates as well.

3.4.5 Hadamard Matrix

Some amount of elementary matrix theory is needed to understand this example.

A family of 0–1 matrices H , known as *Hadamard matrix*, is useful in a variety of applications, particularly for error-correcting codes like in Reed–Muller code. They are defined inductively in Table 3.1 where $\overline{H_n}$ is the bit-wise complement of H_n . Matrices H_1 , H_2 and H_3 are shown in Table 3.2.

Table 3.1 Inductive definition of Hadamard matrix, H_n , $n \geq 0$

$$H_0 = \begin{bmatrix} 1 \end{bmatrix} \quad H_{n+1} = \begin{bmatrix} H_n & H_n \\ H_n & \overline{H_n} \end{bmatrix}$$

Table 3.2 Hadamard matrices, H_1, H_2, H_3

$$\begin{array}{ccc} \left[\begin{array}{c|c} 1 & 1 \\ \hline 1 & 0 \end{array} \right] & \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{array} \right] & \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array} \right] \\ \text{(a) } H_1 & \text{(b) } H_2 & \text{(c) } H_3 \end{array}$$

Hadamard matrices have many pleasing properties. It is easy to see that the number of rows and columns of H_n is 2^n . Also, H_n is symmetric for all n , that is, $H_n = H_n^T$, where H_n^T is the transpose of H_n . This result is easily proved using induction on n .

The property of interest in error-correction is that any two distinct rows of H_n , $n > 0$, differ in exactly half their column positions; so, any pair of rows in H_3 , for example, differ in exactly $2^3/2 = 4$ positions. For binary strings x and y of equal length, let $d(x, y)$ be the number of positions in which they differ; this is known as their *Hamming distance*.

Theorem 3.2 For any two distinct rows x and y of H_n , $n > 0$, $d(x, y) = 2^{n-1}$.

Proof. First I sketch the outline of a direct proof by induction on n . The base case, $n = 1$, follows by inspection. For the inductive case, pick any two rows of H_n , for

$n > 1$. Consider two possible cases: both rows are in (1) the same half, upper or lower half, of the matrix, or (2) different halves of the matrix. For each case, an inductive argument would be separately made.

Next, I show a much simpler proof using problem reformulation.

Proof of Theorem 3.2 using problem reformulation I apply matrix algebra to prove this result. To that end, replace a 0 by -1 and leave a 1 as 1 in the matrix. Then $\overline{H_n} = -H_n$. The scalar product of two rows x and y of H_n , written is written as $x \times y$, and is defined to be

$$(+i : 0 \leq i < 2^n : x_i \cdot y_i).$$

Note that if $x_i = y_i$, then $x_i \cdot y_i = 1$, otherwise, $x_i \cdot y_i = -1$. Therefore, $x \times y = 0$ iff x and y differ in exactly half the positions.

To show that all pairs of distinct rows of H_n differ in exactly half the positions, take the matrix product $H_n \times H_n^T$. Since H_n is symmetric, that is, $H_n = H_n^T$, compute $H_n \times H_n$, and show that the off-diagonal elements, the elements corresponding to the scalar products of distinct rows of H_n , are all zero.

Lemma 3.4 $H_n \times H_n$ is a diagonal matrix, for all $n, n \geq 0$.

Proof. by induction on n .

- $n = 0 : H_0 \times H_0 = \begin{bmatrix} 1 \end{bmatrix} \times \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix}$

- $n + 1 \geq 1 :$

$$\begin{aligned}
 & H_{n+1} \times H_{n+1} \\
 &= \{\text{definition of } H_{n+1}\} \\
 &= \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix} \times \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix} \\
 &= \{\text{matrix multiplication}\} \\
 &= \begin{bmatrix} H_n \times H_n + H_n \times H_n & H_n \times H_n + H_n \times -H_n \\ H_n \times H_n + -H_n \times H_n & H_n \times H_n + -H_n \times -H_n \end{bmatrix} \\
 &= \{\text{simplifying}\} \\
 &= \begin{bmatrix} 2(H_n \times H_n) & 0 \\ 0 & 2(H_n \times H_n) \end{bmatrix}
 \end{aligned}$$

From the induction hypothesis $H_n \times H_n$ is a diagonal matrix, so $2(H_n \times H_n)$ is also a diagonal matrix. Therefore, the matrix shown above is a diagonal matrix. ■

3.4.6 McCarthy's 91 Function

An interesting function devised by John McCarthy requires multiple applications of induction for a proof of one of its properties. McCarthy's 91 function is defined on integers by:

$$f(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ f(f(n + 11)) & \text{if } n \leq 100 \end{cases}$$

It is required to show that for all n , $n \leq 101$, $f(n) = 91$.

Define $rank(n) = 101 - n$, for all n , $n \leq 101$. Proof is by induction on $rank$.

- $rank(n) = 0$: That is, $n = 101$. Then $f(101) = 101 - 10 = 91$.
- $rank(n) > 0$: That is, $n \leq 100$. Then, inductively, that for all m where $rank(m) < rank(n)$, $f(m) = 91$.

Since $rank(n) > 0$, $n \leq 100$ and $f(n) = f(f(n + 11))$. Consider two cases, $n + 11 > 100$ and $n + 11 \leq 100$, for each of which I show that $f(f(n + 11)) = 91$.

Case ($n + 11 > 100$):

$$\begin{aligned} & f(f(n + 11)) \\ = & \{n + 11 > 100; \text{ so, } f(n + 11) = n + 11 - 10 = n + 1\} \\ & f(n + 1) \\ = & \{rank(n + 1) < rank(n); \text{ inductively, } f(n + 1) = 91\} \\ & 91 \end{aligned}$$

Case ($n + 11 \leq 100$):

$$\begin{aligned} & f(f(n + 11)) \\ = & \{n + 11 \leq 100 \Rightarrow rank(n + 11) < rank(n). \text{ Apply induction}\} \\ & f(91) \\ = & \{rank(n) = 101 - n > 100 - n \{n + 11 \leq 100\} \geq 11. \\ & rank(91) = 10. \text{ So, } rank(91) < rank(n). \text{ Apply induction}\} \\ & 91 \end{aligned}$$

■

It follows that f is a total function because for $n > 101$, $f(n) = n - 10$, and for $n \leq 101$, $f(n) = 91$. Also, $f^k(n) = 91$, for all k and n where $k > 0$ and $n \leq 101$.

3.4.7 Topological Order of Partial Orders

Recall the definition of *topological order* of a partial order from Section 2.4.2.1 (page 35). A topological order corresponding to partial order $\preceq$ is a total order $\leq$ such that $x \preceq y$ implies $x \leq y$. The topological order need not be unique.

Every partial order has a topological order. This is a deep theorem that relies on the “Axiom of Choice” of set theory for partial orders over arbitrary infinite sets. Here, I prove the much simpler result for finite sets and countably infinite sets. The proof for finite sets is constructive and is used as the basis for the proof for countably infinite sets. There are more efficient procedures for construction of topological order of finite partial orders, by casting the problem in graph-theoretic terms (see Section 6.6.3, page 303).

3.4.7.1 Topological Order Over Finite Sets

I show an inductive proof for the existence of a topological order $\leq$ corresponding to a partial order $\preceq$ over a finite set S . The proof can be used as the basis of a procedure to construct $\leq$ from $\preceq$.

There is a simple proof of this result using the fact that a finite set has a minimal element x with respect to any partial order; see Exercise 13 of Chapter 2. Inductively, construct a topological order excluding x , and then, let the x be the smallest element in the topological order. I propose a more elaborate proof, excluding an arbitrary element z , so that the proof can be extended to countably infinite sets.

- $S = \{\}$: The topological order is the empty relation.
- $S = T \cup \{z\}$: Inductively, there is a topological order $\leq$ over T that places the elements of T in sequence so that the smaller elements, according to $<$, are to the left and the larger ones are to the right. We have to insert z into this sequence so that all elements smaller than z according to $<$ are to its left and larger than z are to its right.

Define T' to be the set of elements of T so that all elements smaller than z according to $<$ and all elements smaller than them by $<$ are in T' . Formally, for all x and y in T :

$$x \in T' \equiv (\exists y : x \leq y \prec z), \text{ and } T'' = T - T'.$$

Extend $<$ to apply to S : $x < z \equiv x \in T'$, and $z < y \equiv y \in T''$. I show that $\leq$ is a total order over S and for any u and v in S , $u \prec v \Rightarrow u < v$.

- $\leq$ is a total order over S : A pair of elements that does not include z is totally ordered, inductively, in T . And $x < z < y$ iff $x \in T'$ and $y \in T''$, so that all pairs of distinct elements are ordered according to $<$.

- For any u and v in S , $u \prec v \Rightarrow u < v$: If neither of u or v is z , this relationship holds in T inductively. Otherwise, suppose $u \prec z$. The proof for $z \prec v \Rightarrow z < v$ is similar.

$$\begin{aligned}
 & u \prec z \\
 \Rightarrow & \{\text{from } u \leq u \prec z\} \\
 & u \in T' \\
 \Rightarrow & \{\text{definition of } < \text{ with } z\} \\
 & u < z
 \end{aligned}$$

■

Observe an important property of the construction that we use next in constructing topological order over countably infinite sets: if $u \leq v$ in T , then $u \leq v$ in S , so the topological order in T is preserved in S .

3.4.7.2 Topological Order Over Countably Infinite Sets

The inductive proof in Section 3.4.7.1 shows the existence of a topological order for finite sets. The result does not directly extend to infinite sets. I use an important tool for proving facts about infinite structures, “limit construction”, for this proof.

The elements of a countably infinite set S can be indexed with natural numbers (see Section 2.8.5, page 66). Write e_k for the element with index k in S . Let $S_i = \{e_k \mid k \in 0..i\}$, the set of elements with indices below i . Let $\leq_i$ be the topological order over S_i constructed according to the procedure in Section 3.4.7.1. As we noted there, the topological order in S_i is preserved in S_{i+1} , that is, $\leq_i \subseteq \leq_{i+1}$ for all $i, i \geq 0$. Define $\leq = (\cup i : i \geq 0 : \leq_i)$. I show that $\leq$ is a topological order over $(S, \preceq)$.

- $\leq$ corresponds to $\preceq$, $(e_m \preceq e_n) \Rightarrow (e_m \leq e_n)$: Let n be larger than m as a natural number, so both e_m and e_n are in S_n .

$$\begin{aligned}
 & e_m \preceq e_n \\
 \Rightarrow & \{e_m \text{ and } e_n \text{ are in } S_n\} \\
 & e_m \leq_n e_n \\
 \Rightarrow & \{\leq_n \subseteq \leq\} \\
 & e_m \leq e_n
 \end{aligned}$$

- $\leq$ is a total order: That is, I show for any two items e_m and e_n of S , $e_m \leq e_n$ or $e_n \leq e_m$. Let n be larger than m as a natural number, so both e_m and e_n are in S_n .

$$\begin{aligned}
 & \text{both } e_m \text{ and } e_n \text{ are in } S_n \\
 \Rightarrow & \{\leq_n \text{ is a total order over } S_n\} \\
 & e_m \leq_n e_n \text{ or } e_n \leq_n e_m \\
 \Rightarrow & \{\leq_n \subseteq \leq\} \\
 & e_m \leq e_n \text{ or } e_n \leq e_m
 \end{aligned}$$

3.4.8 König's Lemma

As I have explained in Section 3.3.6, induction is not directly applicable to infinite structures such as infinite strings or trees. There are more elaborate mechanisms for applying induction in such cases, as I have shown in constructing the topological order of a countably infinite partial order (see Section 3.4.7.2). Here I prove König's lemma, a result about infinite trees, that is, those having an infinite number of nodes, using induction.

König's lemma is an indispensable tool in reasoning about infinite trees. Sometimes a problem is recast as a problem over infinite trees so that König's lemma can be applied (see Section 3.5.2, page 121, and Exercises 29 and 30, pages 134–135).

Lemma 3.5 König's Lemma

A rooted infinite tree has an infinite path or a node with infinite degree.

Proof. I show an infinite sequence of nodes $\langle r_0, r_1, \dots, r_i, \dots \rangle$ where for each n , $n \geq 0$, (1) the sequence $\langle r_0, r_1, \dots, r_n \rangle$ is a path starting at the root, and (2) r_n is the root of an infinite tree. Proof is by induction on n .

- $n = 0$: Take r_0 to be the root of the given tree.
- $n > 0$: Inductively, $\langle r_0, r_1, \dots, r_{n-1} \rangle$ is a path and r_{n-1} is the root of an infinite tree. Since the degree of r_{n-1} is finite, it has a child r_n that is the root of an infinite tree. So, $\langle r_0, r_1, \dots, r_n \rangle$ is a path starting at the root, and r_n is the root of an infinite tree.

The infinite sequence of nodes $\langle r_0, r_1, \dots, r_i, \dots \rangle$ is an infinite path in the tree. ■

3.4.9 Winning Strategy in Finite 2-Player Game

As an application of König's lemma of Section 3.4.8, I show that a winning strategy exists for a certain class of 2-player games. Consider a 2-player game of strategy, not a physical game like tennis nor a game of chance that involves drawing a card or rolling a dice. The game starts in some initial position, such as the initial board position in chess. The two players alternately make moves out of a finite set of possible moves, and each move transforms the position of the game. The moves and the positions are visible to both players. The game continues until the resulting position is a winning one for one of the players. I restrict the games to those that reach a winning position for one of the players in a finite number of moves, no matter how the players make their moves. This rules out chess, because it may end in a stalemate, or a game like tic-tac-toe, in which the terminal position is usually a draw. An example of a finite 2-player game of this form is Nim.

I show that one of the players has a winning strategy in such a game, that is, either the first or the second player can always make moves so that the terminating position is a win for that player irrespective of how the other player plays. The proof is by induction, and it actually gives the winning strategy, though it is impractical to compute the strategy for most games of interest.

Graphical depiction of finite 2-player game The possible moves of a finite 2-player game can be shown as a tree. Each node of the tree corresponds to a position, though several nodes may correspond to the same position. The root is the initial position. Children of a node correspond to its successor positions.

I illustrate the drawing of a game tree using an example. The game is played with a pile of chips. Each player takes turns removing either two or three chips from the pile. The player who is unable to make a move, because there are fewer than 2 chips in the pile, loses. I show the possible positions and the winning strategy starting with a pile of 10 chips in Figure 3.4. Each node with value n has at most two children with values $n - 2$ and $n - 3$, provided the values are non-negative.

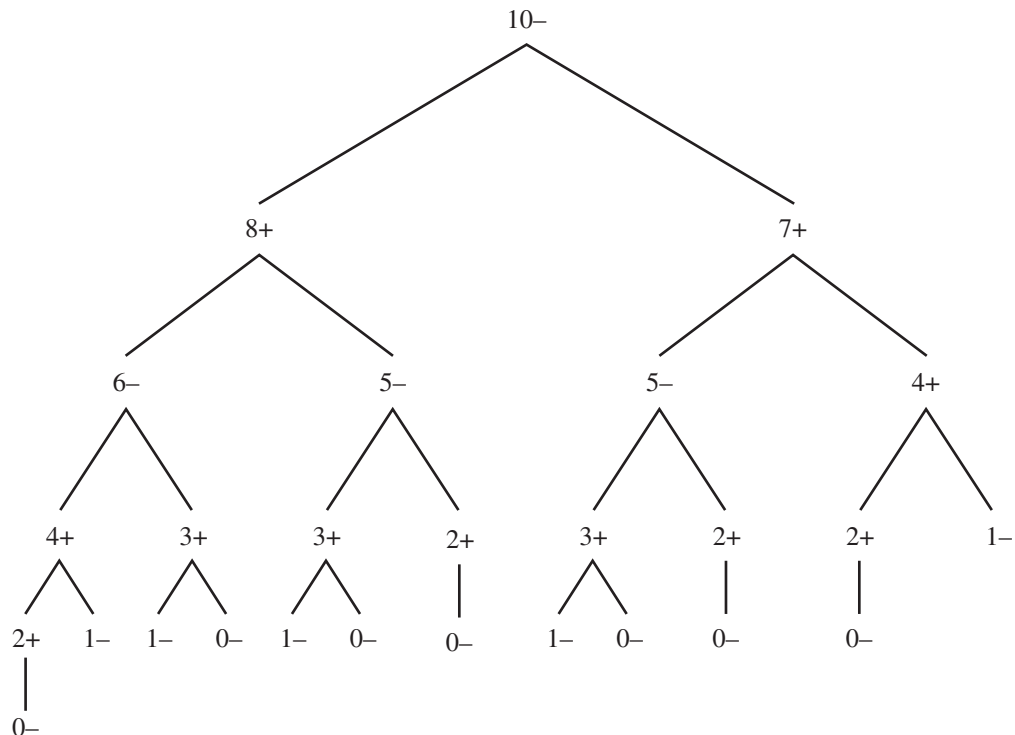


Figure 3.4 Example game tree.

The nodes are also labeled with $+$ or $-$, where symbol $+$ denotes that the player who has to make a move at that position has a winning strategy, $-$ otherwise. Every leaf node is labeled according to the rules of the game: the game is over at that node, so the player who has to make a move loses. Next, if node x has a child y labeled $-$, label x with $+$, and if all children of x are labeled $+$, label x with $-$. I justify this labeling in the proof given below.

In this example, the labeling shows that there is a winning strategy for the second player. Designating the players 1 and 2, I show a possible sequence of moves; here $u \xrightarrow{2} v$ means that in a position with u chips in the pile the second player makes a move that leaves v chips in the pile: $10 \xrightarrow{1} 7 \xrightarrow{2} 5 \xrightarrow{1} 3 \xrightarrow{2} 0$. As expected, the second player wins. The second player, of course, can make a mistake and lose, as in $10 \xrightarrow{1} 7 \xrightarrow{2} 4 \xrightarrow{1} 1$.

Inductive proof of the existence of a winning strategy Recall that all path lengths in a game tree are finite and the degree of a node, that is, the number of possible moves at that position, is also finite. According to [König's lemma](#) of Section 3.4.8, the tree is finite. I show by induction on n that a game tree with at most n nodes has a winning strategy for one of the players.

Call the first player 1 and the second player 2. For $n = 1$, the game tree has a single root node that is also a terminal position, and according to the rules of the game one of the players wins. For $n > 1$ consider the subtrees of the root; each subtree represents a game in which 2 moves first, and each subtree has fewer nodes than n . Inductively, each subtree has a winning strategy for one of the players. If 2 has a winning strategy in all the subtrees, then in the original game 1 can only lose no matter which move he makes. However, if 1 has a winning strategy in one of the subtrees, he can make a move in the original game to the position of the root of that subtree, and he would win from that position.

3.5 Noetherian or Well-founded Induction

Noetherian induction,⁴ also known as *well-founded* induction, generalizes the induction principle over natural numbers. It is more powerful in the sense that it can solve certain problems that cannot be solved using natural-number induction. Moreover, it simplifies reasoning even for cases where natural-number induction is applicable.

It should be clear by now that a proof by induction on natural numbers is actually a compact rendering of an infinite number of proofs that are linearly ordered. Each proof, except the very first one for the base case, may depend on some prior

4. This induction principle is named after its inventor, Emmy Noether.

proofs. Noetherian induction generalizes this idea: instead of a linear order it permits the proofs to be partially ordered. And, instead of a single base case it permits an arbitrary number of base cases.

It is particularly interesting to apply well-founded induction using lexicographic order. As I will show, this allows us to simplify certain proofs that use induction on natural numbers and construct proofs for problems that cannot be solved using natural-number induction. A special form of Noetherian induction, structural induction, can be used to prove facts about recursively defined data structures.

3.5.1 Well-founded Relation

For a binary relation $\prec$ over set W , we say, informally, that x is smaller than y if $x \prec y$. An element m of W is *minimal* in a subset if there is no smaller element than m in that subset. It is not necessary that the minimal element be smaller than all other elements of the subset. Call $(W, \prec)$ *well-founded*, and $\prec$ a *well-founded relation*, if every non-empty subset of W has a minimal element.

There is an equivalent definition of a well-founded set. A *decreasing chain* in $(W, \prec)$ is a sequence of elements of W , $\langle x_0, x_1, \dots, x_i \dots \rangle$, where each element is “smaller” than its predecessor, that is, for all i , $x_{i+1} \prec x_i$. Call $(W, \prec)$ well-founded if it has no infinite decreasing chain.

I show later in this section that the two definitions of well-foundedness are equivalent. It can be shown that the transitive closure of a well-founded relation is a strict partial order, though a strict partial order need not be a well-founded relation.

3.5.1.1 Examples of Well-founded Relations

1. The set of natural numbers is well-founded under the standard “less than”, $<$, relation. The integers are not well-founded under $<$ because the set of integers has no minimal integer. The integers are well-founded under a different order: define $x < y$ iff $|x| < |y|$.
2. Consider a set of finite strings over any alphabet, finite or infinite. The finite strings are well-founded under several different order relations: for strings x and y (1) strict-prefix order where x is a proper prefix of y , (2) strict-suffix order where x is a proper suffix of y , (3) strict-subsequence order where x is a proper subsequence of y , and (4) strict-substring order where x is a contiguous proper subsequence of y . In all cases, any decreasing chain is finite because each string is smaller in length than its predecessor in the chain.

3. A set of finite sets (over elements from some domain) is well-founded under strict-subset order. This is because each proper subset is smaller in size, so there cannot be an infinite decreasing chain.

Note: Infinite sets are not well-founded under strict-subset order. To see this, consider a family of infinite sets of integers where $S_i, i \geq 0$, is the set of integers that are greater than or equal to i . Now $S_{i+1} \subset S_i$, for all i . So, we have the infinite decreasing chain $S_0, S_1, \dots, S_i, \dots$.

4. A set of finite trees are well-founded under strict-subtree order because a proper subtree is smaller in size than the tree.

3.5.1.2 Lexicographic Order

Recall the definition of lexicographic order given in Section 2.4 (page 32). In its simplest form $(x, y) < (x', y')$ if either (1) $(x < x')$ or (2) $(x = x') \wedge (y < y')$, where x and x' , and y and y' belong to the same sets, and $<$ is a total or partial order on those sets. Lexicographic order is a total order when the underlying order is a total order, and a partial order otherwise.

Proposition 3.1 Given that $(S, <)$ is a well-founded set under the total order $<$, $(S \times S, <)$ is well-founded.

Proof. Note that I am overloading $<$ to be a relation over $S \times S$ where $<$ denotes lexicographic order.

I show that there is no infinite decreasing chain in $(S \times S, <)$. For any chain the sequence of contiguous tuples whose first components are identical is called a *segment*. I show that there are a finite number of segments, and each segment has a finite number of pairs, so the chain is finite.

Consider the sequence of first elements of the successive segments (each segment has a unique first element, by construction). For any two successive segments let their adjacent pairs, the last of the upper segment and the first of the lower one, be (x, y) and (x', y') , respectively, so that $(x', y') < (x, y)$. Then $x \neq x'$, from the construction of segments; so, from the definition of $<$, $x' < x$. Therefore, the sequence of first components of successive segments is a decreasing chain. From the well-foundedness of $<$ in S this chain is finite, so the number of segments is finite. And the sequence of second components in any segment forms a chain because the first components are all identical in a segment. This chain is finite because $<$ is well-founded. ■

The order can be extended to finite n -tuples, $n > 1$: define $(x_0, x_1, \dots, x_n) < (y_0, y_1, \dots, y_n)$ if there is some k , $0 \leq k \leq n$, such that $x_k < y_k$ and $x_i = y_i$, for all i , $0 \leq i < k$. Lexicographic order can be extended to Cartesian product of different

sets, each with a different well-founded order, so that for well-founded $(S_1, <_1)$ and $(S_2, <_2)$, $(x, y) < (x', y')$ if $x <_1 x'$, or $x = x'$ and $y <_2 y'$, where x and x' are from S_1 and y and y' from S_2 .

I show a few examples of well-founded lexicographic order; some of these have been discussed in Section 2.4. Standard “less than” relation over positive integers is a well-founded lexical order: to compare 346 against 345 compare their digits successively from left to right until 5 is found to be smaller than 6. For comparing positive integers of unequal lengths, such as 23 against 121, the shorter one is padded with zeroes at the left end to make their lengths equal, so $023 < 121$. For comparisons of words in a dictionary, the shorter word is padded with blanks (white space) at the right end, and blank is taken to be smaller than every other symbol, so that “the” comes before “then”.

In Section 3.4.1 (page 101), we saw that a sequence can be sorted by transposing adjacent items of an inversion. This is true even when the items are non-adjacent. We can prove this result directly, as we did for adjacent transpositions, by showing that it decreases the number of inversions. But it is easier to prove it with lexicographic order. Transpose any pair (x, y) if x precedes y in the sequence and $x > y$. The transposition decreases the entire sequence lexicographically, as in transposing $(1, 3)$ in $\langle 3\ 2\ 7\ 1 \rangle$ yields $\langle 1\ 2\ 7\ 3 \rangle$, a lexicographically smaller sequence. From the well-foundedness of lexicographic order we see that this procedure eventually yields a sorted list.

Lexicographic order applies only to finite tuples, see Exercise 34 (page 135).

3.5.1.3 Equivalence of Two Notions of Well-foundedness

Theorem 3.3 Let W be a set with a binary relation $\prec$. The following statements are equivalent:

- (1) Every non-empty subset of W has a minimal element under $\prec$.
- (2) W has no infinite decreasing chain under $\prec$.

Proof. The proof is by mutual implication.

Suppose there is an infinite decreasing chain. The set of elements in the chain, a subset of W , has no minimal element because every element has some smaller element. Conversely, suppose there is a subset that has no minimal element. Then every element has some smaller element. Then construct an infinite decreasing chain $x_0, x_1, \dots$ where for all $i, i \geq 0, x_{i+1} \prec x_i$. ■

3.5.1.4 Well-founded Induction Principle

The well-founded induction principle is very similar to the strong induction principle over natural numbers from Section 3.1.2. The rewriting of the strong induction principle says that to prove a proposition P for all natural numbers it is sufficient

to prove P for every number assuming that P holds for all smaller natural numbers. That is, if $(\forall i : 0 \leq i < j : P_i) \Rightarrow P_j$, for all $j, j \geq 0$, then P_n holds for all $n, n \geq 0$. Well-founded induction uses a well-founded set $(W, \prec)$ in place of the natural numbers and their standard order.

Well-founded induction principle Given is a well-founded set $(W, \prec)$ and a proposition P . Suppose $(\forall y : y \prec x : P_y) \Rightarrow P_x$, for all $x, x \in W$. Then P_z holds for all $z, z \in W$.

The proof of the principle is by contradiction. Suppose $(\forall y : y \prec x : P_y) \Rightarrow P_x$, for all $x, x \in W$. Let W' be the subset of W consisting of the elements for which P does not hold, that is, $W' = \{z \mid z \in W \wedge \neg P_z\}$. If W' is non-empty, being a subset of W , it has a minimal element m . No element smaller than m is in W' , so P holds for all elements smaller than m . From $(\forall y : y \prec x : P_y) \Rightarrow P_x$, setting x to m , conclude P_m , a contradiction. ■

Induction over natural numbers is a special case of the well-founded induction principle, because the set of natural numbers are well-founded under the standard order: replace $(W, \prec)$ by $(\text{Nat}, <)$, where Nat is the set of natural numbers, to get the strong-induction principle for natural numbers.

3.5.1.5 Examples of Application of Well-founded Induction

Proof of termination of a game A game is played with a coffee can that contains some finite number of black and white beans. The following moves are repeated as long as possible: Randomly select a bean from the can. If it is black, throw it out, and if it is white throw it out, but put any (finite) number of black beans in the can. (Enough extra black beans are available to do this, and the can is large enough.) Does the game always terminate, that is, no bean is left in the can?

It is not immediately obvious that the game terminates because the number of beans in the can may grow without bound. However, we may argue as follows to prove termination. Since there is a finite number of white beans to start with and white beans are never added, there is a point in any run of the game beyond which no white bean is discarded. From that point only black beans are discarded and there is a finite number of black beans in the can. So after a finite number of moves no black bean can be discarded, and since no white bean is discarded the game is over.

The formal treatment of the problem codifies, and simplifies, this reasoning. I claim that the game terminates starting with a can containing w white and b black beans, for any non-negative w and b . Proof is by induction on the set of pairs (w, b) that are well-founded under lexicographic ordering. Assume that the game terminates for all pairs (w', b') that are lexicographically smaller than

(w, b) . Any move with (w, b) decreases the pair lexicographically because the move either decreases w or keeps w the same while decreasing b . Applying well-founded induction, the game terminates from (w, b) .

A generalization of the game Consider the following generalization of the coffee can problem. A game starts with a finite non-empty bag of natural numbers. A *move* replaces an arbitrary positive number j in the bag by any finite number of naturals, possibly repeated, that are strictly smaller than j . Moves are made repeatedly until the bag contains only zeroes. I show that the game always terminates.

Map each bag to a tuple, called its *metric*, such that the set of metrics are well-founded under lexicographic order. I show that each move decreases the metric lexicographically. From the absence of infinite decreasing chains, any game terminates.

First, observe that the maximum value in the bag never increases because each move replaces a number by strictly smaller numbers. Let the maximum value in the initial bag be N . The metric m associated with the bag is a tuple of $N + 1$ components; the i^{th} component of the tuple, m_i , $0 \leq i \leq N$, is the number of occurrences of number i in the bag. The components in a tuple are indexed 0 through N from right to left in the tuple, so the rightmost entry in the tuple is the number of 0s in the bag and the leftmost entry is the number of N s. The set of metric values are well-founded according to lexicographic ordering.

A move replaces some value j in the bag by some number of smaller values, so m_k may change only if $k > j$. Therefore, (1) every m_i , where $i < j$, remains unchanged, and (2) m_j decreases. Hence, m decreases lexicographically with each move.

Ackermann function I prove that the following function, known as the *Ackermann function*, is defined for all values of its arguments. The argument values are restricted to natural numbers.

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

It is not immediately obvious that this definition makes sense because in the last case of the definition the value of the second argument increases.

Proof is by induction on the pairs of natural numbers (m, n) that are ordered lexicographically. For pairs of the form $(0, n)$, the function value is $n + 1$. For all other (m, n) , $m > 0$ and $n \geq 0$, the function value is defined in terms of pairs that are lexicographically smaller: $(m - 1, 1) < (m, n)$ and $(m - 1, A(m, n - 1)) < (m, n)$. Therefore, assuming that A is defined at all smaller values than (m, n) , $A(m, n)$ is

defined. Applying the well-founded induction principle, A is defined for all pairs of natural numbers (m, n) .

3.5.2 Multiset or Bag Ordering

This section contains advanced material.

A subset order over finite sets is well-founded; so the same order also applies for multisets or bags. However, this is not a useful order over bags. An important well-founded order over finite bags was defined by [Dershowitz and Manna \[1979\]](#), which has been used extensively to prove termination in term-rewriting systems.

3.5.2.1 Dershowitz–Manna Order

Let $(D, <)$ be a well-founded set. For finite bags X and Y over D let $A = X - Y$ and $B = Y - X$. Define Dershowitz–Manna order by:

$$Y \prec X \equiv A \neq \emptyset \wedge (\forall y : y \in B : (\exists x : x \in A : y < x)).$$

To understand the definition, observe that Y is constructed from X by removing the elements in A and adding the elements in B . The given order requires that some elements be removed ($A \neq \emptyset$) and each element that is added be smaller than some removed element according to $<$. Henceforth, $X \succ Y$ iff $Y \prec X$.

Example 3.1 Let D be the set of natural numbers with the standard order. Then $\{5, 5, 3, 2\} \succ \{5, 4, 4, 2\} \succ \{5, 4, 3, 3, 3, 1\} \succ \{5, 4\}$. Note that no element is added in the last step though some elements are removed.

3.5.2.2 Dershowitz–Manna Order is a Well-founded Order

I show that any decreasing chain C of bags, $C = X_1 \succ X_2 \dots$, is finite in length. To simplify the proof, postulate a new element $\top$ outside D where $x < \top$ for all $x \in D$. Then $\{\top\} \succ S$ for any bag S over D . Let $X_0 = \{\top\}$ be the first bag in C . The first step removes $\top$ and adds all elements of X_0 .

For any element y that is added to a bag X_{j+1} , $j \geq 0$, there is an element x in X_j that is removed and $y \prec x$. Call x the *parent* of y ; if there are several choices for a parent choose one arbitrarily.

Observe that every element in C , except $\top$, is added at some step because the initial bag, X_0 , has none of the elements of D . So every element in C , except $\top$, has a parent. Construct a tree rooted at $\top$ whose other nodes are the rest of the elements of C , using the parent relation defined above.

The chain C is finite from the following observations.

1. Each path in the tree is a decreasing chain in D . Since D is well-founded each path is finite.

2. Each node in the tree has finite degree: All the children of a node are added to some X_{j+1} in the step in which the node is removed from X_j . Only a finite number of elements are added to a bag in a step. So, each node has a finite number of children.
3. Using König's lemma (Lemma 3.5, page 113) with the above two observations the number of nodes in the tree is finite.
4. The length of C is at most the number of nodes in the tree because each step removes at least one element.

3.6 Structural Induction

Many data structures in computer science are defined recursively. That is, the components of an instance of a structure have the same structure as the original instance. A common example of a recursive data structure is a finite non-empty binary tree, which is either just a single node, its root, or has a root and two binary subtrees. A finite singly linked list is either empty or has a front node followed by a linked list. A finite stack is either empty or has an item on top of a stack. A finite binary string is either empty, or a 0 or 1 followed by a finite binary string. We defined Hadamard matrices recursively in Section 3.4.5. The natural numbers are defined recursively: a natural number is either 0 or 1 plus a natural number. Many other mathematical objects, among them functions, are often defined recursively.

Properties of recursively defined structures can be proved using a particularly convenient form of induction known as *structural induction*, introduced in Burstall [1969]. Informally, to prove property P for every instance of a recursively defined structure, prove P for the base instance(s), and assuming that P holds for the components of an instance, prove P for the instance. Structural induction is based on well-founded induction. The well-founded set is, implicitly, the set of all instances of a given structure, and the order relation, $\prec$, is implicit in that every component of an instance is smaller than the instance. Induction over natural numbers is a very special case of structural induction.

3.6.1 Defining Recursive Structures

There is a variety of ways to define recursive structures: context-free grammars for defining sets of strings, programming language notation for describing recursive functions and abstract grammars for data type definitions. I do not discuss the notational mechanisms for recursive definitions here since our sole aim is to acquaint the reader with the principle of structural induction.

Define the set of instances of structure T by set comprehension, $T = \{x \mid x \in S \text{ and } P(x)\}$. Here S is a set defined previously. Recursive definitions permit the set name T to appear in its own definition, but we have to exercise some caution. First,

T must include some elements, base elements, that are defined without naming T . Next, other elements of T may depend only on “smaller” elements of T , which is implicit in the definition.

As an example, a definition of the set of all finite binary strings is:

$$B = \{\varepsilon\} \cup \{0x \mid x \in B\} \cup \{1x \mid x \in B\}.$$

Here, ε is the empty string, and $0x$ is the string x with an appended 0 at its front; similarly, $1x$. The interpretation of this definition is: a string y is in B iff y is either ε or $y = 0x$ or $y = 1x$ where x is in B . Circularity in the definition is avoided by the requirement that strings are constructed using only a *finite* number of applications of these rules, so y is finite and x is shorter than y . An equivalent interpretation is that B is constructed in stages, starting with ε as its only element, and then adding $0x$ and $1x$ for any x that is already in B . Thus, we have a sequence of sets $\langle B_0, B_1, \dots, B_n, \dots \rangle$, where $B_0 = \{\varepsilon\}$ and $B_{i+1} = \{0x \mid x \in B_i\} \cup \{1x \mid x \in B_i\}$ for all $i \geq 0$, and $B = \bigcup_n B_n$. Observe that B includes only finite binary strings.

I also employ an equivalent and simpler notation where convenient:

$$\begin{aligned} \varepsilon &\in B, \\ x \in B &\Rightarrow 0x \in B, \\ x \in B &\Rightarrow 1x \in B. \end{aligned}$$

This has the intended interpretation. What is assumed implicitly is that nothing else is in B , so B is the smallest set satisfying the given constraints. Such a set always exists if there are base cases and the construction of new elements from the existing elements is well-defined. Previously defined sets may also be used, as before, in a recursive definition. Structural induction is more easily applied with this form of definition, as I illustrate in this section.

Note: It is possible to define a set recursively without a base element, as in:

$$B' = \{0x \mid x \in B'\} \cup \{1x \mid x \in B'\}.$$

This defines B' to be the smallest set that satisfies this equation, the set of infinite binary strings. I do not study such definitions in this book. ■

Examples of recursive set definitions

- The set of natural numbers, *Nat*:

$$\begin{aligned} 0 &\in \text{Nat}, \text{ and} \\ x \in \text{Nat} &\Rightarrow x + 1 \in \text{Nat}. \end{aligned}$$

- The set of positive integers, *Pos*:

$$\begin{aligned} 1 &\in \text{Pos}, \\ x \in \text{Pos} &\Rightarrow 2 \times x \in \text{Pos}, \text{ and} \\ x \in \text{Pos} &\Rightarrow 2 \times x + 1 \in \text{Pos}. \end{aligned}$$

- The set of binary trees BT , written as tuples,⁵ over a given set of nodes, $Nodes$:
 $r \in Nodes \Rightarrow (r) \in BT$,
 $r \in Nodes, s \in BT \text{ and } t \in BT \Rightarrow (s, r, t) \in BT$.

3.6.2 Structural Induction Principle

Consider a generic recursive definition of set S in the form:

(Base) $a \in S$ and $b \in S$, and

(Recursion)

$x_0, x_1 \dots, x_n \in S \Rightarrow f(x_0, x_1 \dots, x_n) \in S$, and

$x_0, x_1 \dots, x_m \in S \Rightarrow g(x_0, x_1 \dots, x_m) \in S$

The base case has two instances, a and b , and there are two recursive definitions.

In practice there could be many more.

To prove proposition P for all elements of S , that is, $(\forall x : x \in S : P(x))$, prove

(Base) $P(a)$ and $P(b)$, and

(Induction)

$(\forall i : 0 \leq i \leq n : P(x_i)) \Rightarrow P(f(x_0, x_1 \dots, x_n))$, and

$(\forall i : 0 \leq i \leq m : P(x_i)) \Rightarrow P(g(x_0, x_1 \dots, x_m))$.

In words, prove proposition P for the base cases, and assuming P for every component x_i , prove P for the structures $f(x_0, x_1 \dots, x_n)$ and $g(x_0, x_1 \dots, x_m)$. There is no need to create an explicit well-founded set and its accompanying order relation.

I justify this scheme by using the induction principle for natural numbers. Let $S_0 = \{a, b\}$ and, for all $i \geq 0$

$S_{i+1} = S_i \cup \{f(x_0, x_1 \dots, x_n) \mid \text{for all } x_j \in S_i\} \cup \{g(x_0, x_1 \dots, x_m) \mid \text{for all } x_j \in S_i\}$.

Then $S = (\cup i : i \geq 0 : S_i)$. Every element x of S can be assigned a *level* j , which is the smallest index such that $x \in S_j$. Now, prove that proposition P holds for all members of $S_j, j \geq 0$, by natural-number induction. This amounts to proving that P holds for all elements of S_0 , which is just the base case above, and for any $y \in S_{i+1}$ that $(\forall x : x \in S_i : P(x)) \Rightarrow P(y)$, which follows from the induction proof step in the structural induction principle.

Functions defined on recursive structures Recursive definitions of data structures permits recursive definitions of functions on those structures. I devote Chapter 7 to recursive programming and apply it extensively in Chapter 8. Here I show an example of a function defined on a recursive structure, and prove a property of the function.

5. I permit a single element tuple in this example. In the rest of the book, tuples have more than one element.

The set of binary trees, BT , is defined in Section 3.6.1 (page 124) over a given set of nodes, $Nodes$:

$$\begin{aligned} r \in Nodes &\Rightarrow (r) \in BT, \\ (r \in Nodes, s \in BT, t \in BT) &\Rightarrow (s, r, t) \in BT. \end{aligned}$$

Define the number of vertices, nv , and edges, ne , of a tree:

$$\begin{aligned} nv((r)) &= 1, nv((s, r, t)) = 1 + nv(s) + nv(t), \text{ and} \\ ne((r)) &= 0, ne((s, r, t)) = 2 + ne(s) + ne(t). \end{aligned}$$

These definitions can be justified. A tree with a single node, (r) , has one vertex and no edge. The number of vertices in (s, r, t) , with root r and subtrees s and t , is the number of vertices in s plus the number in t and 1 more for vertex r . Similarly, its number of edges is the number in s plus the number in t and 2 more for the edges (r, s) and (r, t) .

I prove that the number of vertices is 1 more than the number of edges in a binary tree. Proof is by structural induction.

(Base) Show $nv((r)) = 1 + ne((r))$: from the definition, $1 = 1 + 0$.

(Induction) Show $nv((s, r, t)) = 1 + ne((s, r, t))$:

$$\begin{aligned} &nv((s, r, t)) \\ &= \{\text{definition}\} \\ &\quad 1 + nv(s) + nv(t) \\ &= \{\text{inductive hypothesis}\} \\ &\quad 1 + (1 + ne(s)) + (1 + ne(t)) \\ &= \{\text{rewriting}\} \\ &\quad 1 + (2 + ne(s) + ne(t)) \\ &= \{\text{definition}\} \\ &\quad 1 + ne((s, r, t)) \end{aligned}$$

3.7 Exercises

1. Prove the following identities. Below n and i are natural numbers.

$$(a) (+i : 0 \leq i \leq n : i^2) = \frac{n \times (n+1) \times (2n+1)}{6}.$$

$$(b) (+i : 0 \leq i \leq n : i \times (i+1)) = \frac{n \times (n+1) \times (n+2)}{3}.$$

$$(c) (+i : 0 \leq i \leq n : i^3) = \frac{n^2 \times (n+1)^2}{4}.$$

$$(d) (+i : 1 \leq i \leq n : \frac{1}{i \times (i+1)}) = \frac{n}{n+1}.$$

$$(e) (+i : 0 \leq i \leq n : (-1)^i \times i^2) = (-1)^n \times n \times (n+1)/2.$$

$$(f) (+i : 0 \leq i \leq n : r^i) = \frac{r^{n+1}-1}{r-1}, r \text{ any real number, } r \neq 1.$$

$$(g) k! > 2^k, \text{ for any integer } k, k \geq 4.$$

2. Prove the following propositions where n is any natural number.
- (a) $13^n + 6$ is divisible by 7, for even n .
 - (b) $4^{n+2} + 5^{2n+1}$ is divisible by 21.
 - (c) $n^3 - n$ is divisible by 3.
 - (d) Sum of the cubes of any three consecutive integers is divisible by 9.
 - (e) For $n > 7$, $n = 3 \times a + 5 \times b$, for some integers a and b .
 - (f) $(1 + x)^n \geq 1 + n \cdot x$, for any real number x provided $1 + x \geq 0$.
3. Let $x_0 = 1$, and $x_{n+1} = x_n + 1/x_n$, for $n \geq 0$. Show that for all natural n , $n \geq 1$, $x_n > +\sqrt{n}$.

Solution Show the equivalent result, $x_n^2 > n$ and $x_n > 0$. Prove the base case, $x_1^2 > 1$, from $x_1 = x_0 + 1/x_0$. Proof of $x_{n+1}^2 > n + 1$ for $n > 1$:

$$\begin{aligned}
 & x_{n+1}^2 \\
 & > \{ \text{inductively, } x_n > \sqrt{n}. \text{ Hence, } x_n > 0. \\
 & \quad x_{n+1} = x_n + 1/x_n; \text{ so } x_{n+1} > x_n > 0 \} \\
 & \quad x_{n+1} \times x_n \\
 & = \{ \text{expanding } x_{n+1} \} \\
 & \quad (x_n + 1/x_n) \times x_n \\
 & = \{ \text{algebra} \} \\
 & \quad x_n^2 + 1 \\
 & > \{ \text{induction hypothesis} \} \\
 & \quad n + 1
 \end{aligned}$$

Also, $x_{n+1} > x_n > 0$ has been shown during the proof.

4. (Fibonacci sequence; problem ascribed to Hemachandra) Let h_n be the number of ways a string of symbols of length n can be partitioned into pieces (substrings) of length 1 or 2. So, $h_0 = 0$ because no such partition is possible, $h_1 = 1$ because a string of length 1 can be partitioned in only one way, $h_2 = 2$ because there could be two pieces of length 1 or one of length 2, and a string of length 3 can be divided into substrings of lengths (1, 1, 1), (1, 2), or (2, 1). What is h_n ?
5. Prove the following facts where F_n is the n^{th} Fibonacci number, for $n \geq 0$.
- (a) $F_n < 2^n$, for all n , $n \geq 0$.
 - (b) $\gcd(F_n, F_{n+1}) = 1$, for all n , $n \geq 0$.
 - (c) $(+i : 0 \leq i \leq n : F_i^2) = F_n \times F_{n+1}$, $n \geq 0$.

- (d) $F_{m+n+1} = F_m \times F_n + F_{m+1} \times F_{n+1}$, for all m and n , $m \geq 0$ and $n \geq 0$.
- (e) F_m divides $F_{m \cdot n}$, for all m and n , $m > 0$ and $n \geq 0$.
- (f) $F_n > \phi^{n-2}$, for all n , $n \geq 3$, where $\phi = \frac{1+\sqrt{5}}{2}$.

Hint Use induction on n in all cases. Use the result of item (5d) in proving item (5e). For proving item (5f), use $\phi^2 = 1 + \phi$.

6. There is a neat way to derive many facts about Fibonacci numbers using matrix algebra. Below, the determinant of matrix A is written as $\|A\|$. The following facts about determinants are needed for solving this exercise.

(i). For matrices A and B where $A \times B$ is defined, $\|A \times B\| = \|A\| \times \|B\|$.

(ii). The determinant of

$$\begin{bmatrix} a & b \\ d & c \end{bmatrix} \text{ is } (a \times c - b \times d).$$

Next, define matrix Q whose elements are Fibonacci numbers, F_0 , F_1 and F_2 :

$$Q = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

(a) Prove that for all n , $n \geq 1$,

$$Q^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

(b) Show that $\|Q^n\| = (-1)^n$, for all n , $n \geq 1$.

(c) Show (this part does not need induction)

$$\begin{aligned} & \begin{bmatrix} F_{m+1} \times F_{n+1} + F_m \times F_n & F_m \times F_{n+1} + F_{m-1} \times F_n \\ F_{m+1} \times F_n + F_m \times F_{n-1} & F_m \times F_n + F_{m-1} \times F_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} F_{m+n+1} & F_{m+n} \\ F_{m+n} & F_{m+n-1} \end{bmatrix} \end{aligned}$$

Solution

- (a) Apply induction on n .
- (b) From the facts about determinants, (i), show $\|Q^n\| = \|Q\|^n$. And from (ii) $\|Q\| = -1$.
- (c) Use $Q^m \times Q^n = Q^{m+n}$. Note that the corresponding entries of the two matrices are equal, which seemingly gives four identities. But they are covered by the single identity in Exercise 5d (page 127), by reversing

the roles of m and n , and replacing m and n by $m - 1$ and $n - 1$ in some of the identities.

7. Harmonic numbers h are defined in Section 3.2.1.4 (page 89). Show that $(+i : 1 \leq i \leq n : h_i) = (n + 1) \cdot h_n - n$, for all positive integers n .

Solution

(a) $n = 1$: The left side is $h_1 = 1$. The right side is $(1 + 1) \cdot h_1 - 1 = 1$.

(b) $n > 1$:

$$\begin{aligned}
 & (+i : 1 \leq i \leq n : h_i) \\
 = & \{\text{arithmetic}\} \\
 & (+i : 1 \leq i \leq n - 1 : h_i) + h_n \\
 = & \{\text{induction hypothesis}\} \\
 & n \cdot h_{n-1} - (n - 1) + h_n \\
 = & \{\text{arithmetic}\} \\
 & n \cdot (h_{n-1} + 1/n) - n + h_n \\
 = & \{\text{definition of harmonic numbers}\} \\
 & n \cdot h_n - n + h_n \\
 = & \{\text{arithmetic}\} \\
 & (n + 1) \cdot h_n - n
 \end{aligned}$$

8. Let x and y be real numbers. Define $f_0(x, y) = (x, y)$ and for any natural i , $f_{i+1}(x, y) = f_i((x+y)/2, (x-y)/2)$. Show that $f_{2^n}(x, y) = (x/2^n, y/2^n)$, for all natural n .

9. For distinct integers x, y and natural n , show that $(x - y)$ divides $(x^n - y^n)$.

Hint Write $(x^{n+1} - y^{n+1})$ as $x^{n+1} - xy^n + xy^n - y^{n+1}$. Factor the first two terms and the last two.

10. Show $(1 + z)^n = (+i : 0 \leq i \leq n : \binom{n}{i} z^i)$ using induction on n .

Hint Assume the identity $\binom{n+1}{i} = \binom{n}{i} + \binom{n}{i-1}$, where $\binom{n}{i} = 0$ for $i < 0$ and $i > n$.

11. Show that for $n > 1$ the following expression is irrational, where each square root yields a positive value.

$$\underbrace{\sqrt{1 + \sqrt{1 + \dots}}}_{n \text{ 1s}}$$

12. Let $X = \langle x_1, x_2, \dots \rangle$ be an infinite sequence where $x_{i+1} = f(x_i)$ for some given function f , for all i . Suppose that there are identical values in the sequence, that is, for some m and p , $x_m = x_{m+p}$, where $m \geq 1$ and $p > 0$. Show that for all i and k , where $i \geq m$ and $k > 0$, $x_i = x_{i+kp}$.

Hint First show that $x_i = x_{i+p}$ for all $i, i \geq m$, by induction on i . Next, show that $x_i = x_{i+kp}$ by induction on k .

13. De Morgan's Law was described as follows in Section 2.5.2:

$$\neg(p \wedge q) = (\neg p \vee \neg q), \text{ and}$$

$$\neg(p \vee q) = (\neg p \wedge \neg q)$$

Prove the following generalizations for $n \geq 0$ using the associativity of $\wedge$ and $\vee$:

$$\neg(\wedge i : 0 \leq i \leq n : p_i) = (\vee i : 0 \leq i \leq n : \neg p_i).$$

$$\neg(\vee i : 0 \leq i \leq n : p_i) = (\wedge i : 0 \leq i \leq n : \neg p_i).$$

14. Construct trimino tilings for $2^n \times 2^n$ boards where n is 1, 2 and 3. Choose an arbitrary closed cell in each case.

15. Why can't the induction hypothesis be used, instead of Rule (R1), in the last step of the proof of Lemma 3.1 (page 92)?

16. We saw in Section 3.4.1 (page 101) that any sequence can be permuted arbitrarily by using transpositions. Why can't this result be used to prove $b^n \circ w^n \sim w^n \circ b^n$ in the pebble movement game of Section 3.2.2.2 (page 92)?

Solution Arbitrary transpositions of items are not allowed in the pebble movement game. Only after proving Lemma 3.2 (page 92), we know that adjacent transpositions are valid, so any permutation can be realized.

17. (Pseudo-ternary number representation) A pseudo-ternary representation of natural number n is a non-empty string of coefficients $\langle c_m, \dots, c_0 \rangle$ where each c_i is $-1, 0$ or 1 and $n = (\sum i : 0 \leq i \leq m : c_i \times 2^i)$. For example, number 13 can be represented by $(1, 0, 0, -1, -1)$, $(1, 1, 0, 1)$ or $(1, 0, -1, 0, 1)$. The size of a representation is the number of non-zero entries in it. An optimal representation has the smallest size, call it the optimal size. Each of the given representations for 13 is optimal.

Denoting the optimal size of n by $\bar{n}$, show that:

$$\text{even}(n) \Rightarrow \bar{n} \leq \overline{n+1} \leq \bar{n} + 1 \quad (\text{P1})$$

$$\text{odd}(n) \Rightarrow \overline{n+1} \leq \bar{n} \leq \overline{n+1} + 1 \quad (\text{P2})$$

Solution First, observe that in any representation of n the lowest coefficient $c_0 = 0$ iff n is even, which follows from $n = (\sum i : 0 \leq i \leq m : c_i \times 2^i)$. Therefore, an optimal representation of an even number $2 \cdot t$ is the optimal representation of t appended with 0 at the end. So,

$$\overline{2 \cdot t} = \bar{t} \quad (1)$$

Any representation of an odd number $2 \cdot t + 1$ has either a $+1$ or a -1 as its last coefficient corresponding to the representation of $2 \cdot t + 1$ as $(2 \cdot t) + 1$ or $2 \cdot (t + 1) - 1$, and the optimal representation is one that has the smaller size. Hence,

$$\overline{2 \cdot t + 1} = \min(\overline{t}, \overline{t + 1}) + 1 \quad (2)$$

To prove the claims (P1) and (P2), apply induction on the magnitude of n . Show that $\overline{0} = 0$, $\overline{1} = 1$, $\overline{2} = 1$, which proves the result for the smallest even and odd values of n . Next, consider n , $n \geq 2$. I prove the result when n is even, the proof for odd n is similar. Given that n is even, we have to show, from (P1), $\overline{n} \leq \overline{n + 1} \leq \overline{n} + 1$. Letting $n = 2 \cdot t$, this is $\overline{2 \cdot t} \leq \overline{2 \cdot t + 1} \leq \overline{2 \cdot t} + 1$. Using equations (1) and (2), we need to show:

$$\overline{t} \leq \min(\overline{t}, \overline{t + 1}) + 1 \leq \overline{t} + 1. \quad (3)$$

Consider two cases, $\overline{t} < \overline{t + 1}$ and $\overline{t + 1} \leq \overline{t}$.

• $\overline{t} < \overline{t + 1}$: Then $\min(\overline{t}, \overline{t + 1}) + 1 = \overline{t} + 1$ and (3) follows.

• $\overline{t + 1} \leq \overline{t}$: Then $\min(\overline{t}, \overline{t + 1}) + 1 = \overline{t + 1} + 1$. Inductively, $\overline{t}$ and $\overline{t + 1}$ differ by at most 1; so $\overline{t} \leq \overline{t + 1} + 1 \leq \{\overline{t + 1} \leq \overline{t} \text{ from the assumption}\} \overline{t} + 1$.

18. Design an algorithm to compute the optimal pseudo-ternary representation of any natural number, as described in Exercise 17.

Solution from the solution to Exercise 17, for an even number $2 \cdot t$ compute the optimal representation of t and append 0 at the end. For odd number $2 \cdot t + 1$, compute the optimal representations of t and $t + 1$, choose the one of smaller size and append either 1 or -1 , as appropriate, at its end. Observe that the set of numbers whose optimal representations will be computed is at most $2 \times \log_2 n$.

19. For integers n and k , $\binom{n}{k}$ is the binomial coefficient. Write $\langle \binom{n}{k} \rangle$ for the lowest bit in the binary expansion of $\binom{n}{k}$. Prove that for n , k and any natural number t ,

$$\langle \binom{n+2^t}{k} \rangle = \langle \binom{n}{k} \rangle \oplus \langle \binom{n}{k-2^t} \rangle, \text{ where } \oplus \text{ is addition modulo 2.}$$

Hint Apply induction on t . Assume the identity

$$\binom{n+1}{i} = \binom{n}{i} + \binom{n}{i-1}, \text{ where } \binom{n}{i} = 0 \text{ for } i < 0 \text{ or } i > n.$$

Note There is a stronger theorem by Lucas:

$$\binom{n}{k} \equiv \binom{n_0}{k_0} \times \binom{n_1}{k_1} \cdots \times \binom{n_i}{k_i},$$

where p is any prime, and n_i s and k_i s are the digits of n and k in their p -ary expansions with n_0 and k_0 being the leading digits. With $p = 2$, this

says that $\binom{n}{k}$ is odd iff each $\binom{n_i}{k_i}$ is 1, for all i , where n_i and k_i are the i^{th} bits in the corresponding binary expansions. That is, $n_i \geq k_i$, or equivalently, $k_i = 1 \Rightarrow n_i = 1$.

20. Write $m \mid n$ to mean that m divides n . Show that for all natural numbers p , $3^{p+1} \mid (2^{3^p} + 1)$.

Solution Proof is by induction on p . For $p = 0$, $3 \mid (2^1 + 1)$. For $p + 1$ where $p \geq 0$ we have to show $3^{p+2} \mid (2^{3^{p+1}} + 1)$. Abbreviate 2^{3^p} by x . Then, $2^{3^{p+1}} = 2^{3^p \times 3} = (2^{3^p})^3 = x^3$. Inductively, $3^{p+1} \mid (x + 1)$.

$$\begin{aligned}
 & 3^{p+1} \mid (x + 1) \\
 \Rightarrow & \{p \geq 0\} \\
 & 3 \mid (x + 1) \\
 \Rightarrow & \{\text{arithmetic}\} \\
 & 3 \mid ((x + 1)^2 - 3 \times x) \\
 \equiv & \{\text{arithmetic}\} \\
 & 3 \mid (x^2 - x + 1) \\
 \Rightarrow & \{\text{inductive hypothesis: } 3^{p+1} \mid (x + 1)\} \\
 & (3 \times 3^{p+1}) \mid ((x + 1) \times (x^2 - x + 1)) \\
 \Rightarrow & \{\text{algebra on both terms}\} \\
 & (3^{p+2}) \mid (x^3 + 1) \\
 \Rightarrow & \{x = 2^{3^p}\} \\
 & 3^{p+2} \mid (2^{3^{p+1}} + 1)
 \end{aligned}$$

21. (Generalization of the previous exercise) Show that $3^{p+1} \mid (2^q + 1)$, where $q = 3^{p(2 \times t + 1)}$ and p and t are natural numbers.

Hint Solution is similar to the previous one. Here 2^q plays the role of x . Increasing p by 1 gives q^3 .

22. (Hard exercise) Let f be a function from naturals to naturals such that $f(f(n)) < f(n + 1)$, for all naturals n . Show that f is the identity function.

Hint Prove the result in the following steps.

- (a) f is at least its argument value, that is, $n \leq f(n)$, for all n .
- (b) $f(n) < f(m) \Rightarrow n < m$, for all m and n .
- (c) f is the identity function, that is, $f(n) = n$, for all n .

Solution

- (a) I prove a more general result, that for all t and n , $t \geq 0$ and $n \geq 0$, $n \leq f(n + t)$. The desired result follows by setting t to 0. Proof is by induction on n .

(Base) $0 \leq f(t)$, for all $t, t \geq 0$, because f is a function from naturals to naturals.

(Induction): I prove that $n + 1 \leq f(n + 1 + t)$ for all t , assuming that $n \leq f(n + s)$ for all $s, s \geq 0$.

$$\begin{aligned}
 & \text{true} \\
 \Rightarrow & \{\text{Induction hypothesis}\} \\
 & n \leq f(n + s), \text{ for all } s, s \geq 0 \\
 \Rightarrow & \{\text{setting } s \text{ to } t \text{ above: } n \leq f(n + t), \text{ so } f(n + t) - n \geq 0. \\
 & \text{Set } s \text{ to } f(n + t) - n \text{ in } n \leq f(n + s).\} \\
 & n \leq f(n + f(n + t) - n) \\
 \Rightarrow & \{\text{Rewriting}\} \\
 & n \leq f(f(n + t)) \\
 \Rightarrow & \{\text{From } f(f(n)) < f(n + 1), \text{ setting } n \text{ to } n + t: f(f(n + t)) < \\
 & f(n + 1 + t)\} \\
 & n < f(n + 1 + t) \\
 \Rightarrow & \{\text{arithmetic}\} \\
 & n + 1 \leq f(n + 1 + t)
 \end{aligned}$$

(b) Setting n to $f(n)$ in item (a) above, $f(n) \leq f(f(n))$. Using $f(f(n)) < f(n + 1)$ conclude $f(n) < f(n + 1)$. Then $m \leq n \Rightarrow f(m) \leq f(n)$. Take the contrapositive to get the desired result.

(c) Given $f(f(n)) < f(n + 1)$,

$$\begin{aligned}
 & f(f(n)) < f(n + 1) \\
 \Rightarrow & \{\text{from item (b)}\} \\
 & f(n) < n + 1 \\
 \Rightarrow & \{\text{from item (a): } n \leq f(n)\} \\
 & n \leq f(n) < n + 1 \\
 \Rightarrow & \{f \text{ is a function to naturals}\} \\
 & n = f(n)
 \end{aligned}$$

23. (Three-halves conjecture) A famous open problem in mathematics, known as the three-halves conjecture, asks for a proof of termination of the following iterative procedure for any positive integer n . If n is odd then set n to $(3 \cdot n + 1)/2$, and if n is even set it to $n/2$. Repeat the steps until n becomes 1. This conjecture has been verified for many values of n . However, there is no proof yet for all n .

Show that if this procedure terminates for t , then it terminates for $2^p \cdot (t + 1)/3^p - 1$, for every $p, p \geq 0$, provided the given expression denotes a positive integer.

Hint First prove that if $2^p \cdot (t + 1)/3^p - 1$ is integer for any $p, p \geq 1$, then it is odd. Next, prove the required result by induction on p .

24. (Three-halves conjecture, contd.) In the three-halves conjecture an n -upchain, for $n > 0$, is a sequence $\langle x_0, x_1, \dots, x_{n-1} \rangle$ of distinct odd integers where $x_{i+1} = (3 \cdot x_i + 1)/2$ for all $i, 0 \leq i < n - 1$. Show that $x_0 \geq 2^n - 1$. It follows from this result that there is no infinite upchain.

Solution Apply induction on length n of the upchain $\langle x_0, x_1, \dots, x_{n-1} \rangle$.

For $n = 1$: $x_0 \geq 2^1 - 1 = 1$ because all elements of an upchain are odd.

For $n + 1, n \geq 1$: Each $x_i, 0 \leq i \leq n$, is odd and $x_i \geq 3$ because any upchain starting at 1 has length 1. Define a sequence of positive integers $y : \langle y_0, y_1, \dots, y_{n-1} \rangle$, for $0 \leq i \leq n$, where $x_i = 2 \cdot y_i + 1$. For any two consecutive items s and t in y the two corresponding items in x are $2 \cdot s + 1$ and $2 \cdot t + 1$. Therefore, $2 \cdot t + 1 = (3 \cdot (2 \cdot s + 1) + 1)/2$, or $t = (3 \cdot s + 1)/2$. In order for t to be integer, s is odd. Therefore, every item of y , except possibly y_n , is odd and $y_{i+1} = (3 \cdot y_i + 1)/2$. So, $\langle y_0, y_1, \dots, y_{n-1} \rangle$ is an n -upchain.

Applying induction on n -upchain $\langle y_0, y_1, \dots, y_{n-1} \rangle$, $y_0 \geq 2^n - 1$, or $x_0 = 2 \cdot y_0 + 1 \geq 2^{n+1} - 1$.

25. Show that the sum of the interior angles of a convex n -gon is $180 \times (n - 2)$ degrees.
26. A *full* binary tree is a binary tree in which every internal node has exactly two children. Let b be the bag of path lengths to the all the leaves from the root of the tree. Show that $(+i : i \in b : 2^{-i}) = 1$.

Hint Use induction on the size of b .

27. A couple (called host and hostess) invites four other couples to a party. Each person shakes some hands, possibly 0, but no one shakes hands with his/her spouse or his/her self. The host then determines that each of the others have shaken distinct numbers of hands. How many hands did the hostess shake?

Solution Generalize the problem to n invited couples. So, the set of hands shaken by all except the host is $\{i \mid 0 \leq i \leq 2 \cdot n\}$ because the minimum number of shaken hands is 0, the maximum number is $2 \cdot n$ discounting the person and his/her spouse, and there are exactly $2 \cdot n + 1$ persons in the party besides the host.

Let p_i be the person other than the host who shakes i hands, $0 \leq i \leq 2 \cdot n$. It is given that all p_i s are distinct. I prove the proposition that p_t and $p_{2 \cdot n - t}$ are spouses, for all t , $0 \leq t < n$. For $n = 0$, the proposition holds vacuously because the range $0..n$ is empty. For any n , $n > 0$, the only person with whom $p_{2 \cdot n}$ does not shake hand is p_0 because p_0 shakes no hands and $p_{2 \cdot n}$ shakes hands with all others; so p_0 and p_n are spouses because $p_{2 \cdot n}$ shakes all possible hands. Removing p_0 and $p_{2 \cdot n}$ reduces every other person's hand-shake count by 1 because $p_{2 \cdot n}$ shakes hands with all of them and p_0 shakes no hand. Therefore, the situation is identical with the remaining $n - 1$ couples. Inductively, p_t and $p_{2 \cdot (n-1) - t}$ are spouses, for all t , $0 \leq t < n - 1$. Thus, the proposition is proved for all n .

Observe that except for p_n every person has a spouse in this list of p s. Since the hostess's spouse, the host, is none of the p_i s, the hostess is p_n ; that is, she shakes n hands.

28. Consider the equation $au = ub$ where a and b are symbols and u is a finite string of symbols. Show that $a = b$ and all symbols in u are this common symbol.
29. A sequence of real numbers $\langle x_0, x_1, x_2, \dots \rangle$ is *ascending* if $x_0 \leq x_1 \leq x_2 \leq \dots$, and *descending* if $x_0 \geq x_1 \geq x_2 \geq \dots$. Call a subsequence monotone if it is either ascending or descending. Show that every infinite sequence of real numbers has a monotone infinite subsequence.

Solution Let the given infinite sequence be S and its elements $S_1, S_2, \dots$. Let S_0 have the value $-\infty$, a fictitious value smaller than every real number. Construct an infinite tree from S whose nodes are labeled with distinct natural numbers. The root is labeled 0 and the parent of any other node j is i provided S_i is the closest previous value to S_j that is smaller than S_j . So, if i is the parent of j , then $i < j$, $S_i < S_j$ and for all k , $i < k < j$, $S_k \geq S_j$. Parent is well-defined for every node other than the root, because S_0 is smaller than every other value.

Consider children q and r of a node p , so $p < q$ and $p < r$. If $q < r$, then $S_q \geq S_r$, because from $p < q < r$, every intermediate value between p and r , in particular S_q , is greater than or equal to S_r . So, the children of a node define a descending sequence. By construction, every path is an increasing sequence. Using König's lemma, the tree has an infinite path or a node with infinite degree. Therefore, S has an infinite increasing, hence ascending, sequence in the first case and an infinite descending sequence in the second case.

Another solution A related problem for finite sequences has been solved in Section 2.2.7.1 (page 24). It shows that every finite sequence whose length exceeds n^2 has a monotone subsequence of length greater than n . Therefore, there is no bound for the longest monotone subsequence in an infinite sequence.

30. Prove Proposition 3.1 (page 117) using König's lemma.

Solution The proof is similar to that of Exercise 29. I construct a tree whose nodes are from the given decreasing sequence, and, using König's lemma, prove that the tree is finite and so is the decreasing sequence.

Postulate an element $\top$ that is larger than any element of S . Let C be a decreasing chain; without loss in generality let $(\top, \top)$ be the first element of C . Construct a tree out of C using the following rules: (1) the root of the tree is $(\top, \top)$, (2) for any other pair (x, y) in C its parent is the closest preceding pair (x', y') such that $x' > x$. Every pair has a parent because of the existence of $(\top, \top)$ in S .

It follows from the construction that every path is a subsequence of C , and a decreasing chain in S considering the first components only. Further, any two children of node (x', y') , (x, y) and (z, w) , have $x = z$. This is because if $x > z$, then (x, y) would be a closer preceding pair than (x', y') whose first component is larger than x . Therefore, the children of each node form a subsequence of C and a decreasing chain in S . Given that $(S, <)$ is well-founded, each path length and degree of each node is finite. According to König's lemma, the tree is finite, and so is C .

31. Show that if relation $\prec'$ is a subset of relation $\prec$ and $(W, \prec)$ is well-founded, then $(W, \prec')$ is well-founded. In particular, if $\prec'$ is the empty relation, then $(W, \prec')$ is well-founded.
32. Show that the transitive closure of a well-founded relation is a strict partial order. And show that a strict partial order need not be a well-founded relation.
33. Let $(W, \prec)$ be well-founded. Suppose $\prec'$ is a binary relation over W' and f a function, $f : W' \rightarrow W$, so that $(x \prec' y) \Rightarrow (f(x) \prec f(y))$. Show that $(W', \prec')$ is well-founded.

Hint Any chain in W' maps to a chain in W .

34. Show an example of tuples with infinite number of components that are not well-founded under lexicographic order.

Solution Let the i^{th} tuple have i leading zeroes followed by infinite number of 1s, for all $i, i \geq 0$. Using $0 \prec 1$, the $(i+1)^{\text{th}}$ tuple is smaller than the i^{th} tuple, for all i . Thus, there is an infinite decreasing chain.

35. For Ackermann function A from Section 3.5.1.5 (page 120) and natural m and n , show:

- (a) $A(1, n) = n + 2$,
- (b) $A(2, n) = 2n + 3$,
- (c) $A(n, n) \geq 2^n, n \geq 0$, and
- (d) $A(m, n) < A(m, n + 1) \leq A(m + 1, n)$.

36. Show that set Pos , defined in Section 3.6.1 (page 123), contains every positive integer and nothing else.

37. Let S be a set of real numbers given by:

$$0 \in S, 1 \in S,$$

$$x \in S \wedge y \in S \Rightarrow (x + y)/2 \in S.$$

Prove that $1/3 \notin S$.

Hint Find an inductive proposition P that holds for every element of S and does not hold for $1/3$.

Solution Use the following inductive proposition:

$$P(x) \in S \Rightarrow (\exists m, n : m \in \text{Nat}, n \in \text{Nat} : x = m/2^n).$$

38. The following algorithm determines if a positive integer represented in decimal notation is divisible by 11. Start from the right end of the number alternately adding and subtracting the digits; for example, 154 gives $4 - 5 + 1$, that is, 0. The number is divisible by 11 iff the computed result is 0. Prove the correctness of this procedure.

Solution Let $digit$ be the set of digits from 0 through 9. Use structural induction to define $posdec$, the set of positive integers written in decimal. For $d \in digit$ and $n \in posdec$: $d \in posdec$ and $(10 \times n + d) \in posdec$.

Let function $asum$ applied to an element of $posdec$ give the result of alternate addition and subtraction as described above. Formally:

$$asum(d) = d, \text{ and } asum(10 \times n + d) = d - asum(n).$$

For any element m of $posdec$, I show that $m = asum(m) \pmod{11}$. The desired result follows as a corollary.

For $m \in digit$, the identity is easy to see since $m = asum(m)$. For the general case, let $m = 10 \times n + d$.

$$\begin{aligned}
& \text{asum}(m) \bmod 11 \\
= & \{m = 10 \times n + d\} \\
& \text{asum}(10 \times n + d) \bmod 11 \\
= & \{\text{definition of asum}\} \\
& (d - \text{asum}(n)) \bmod 11 \\
= & \{\text{properties of mod}\} \\
& (d \bmod 11 - \text{asum}(n) \bmod 11) \bmod 11 \\
= & \{n < m, \text{ apply induction hypothesis}\} \\
& (d \bmod 11 - n \bmod 11) \bmod 11 \\
= & \{\text{properties of mod}\} \\
& (10 \times n \bmod 11 + d \bmod 11) \bmod 11 \\
= & \{\text{properties of mod}\} \\
& (10 \times n + d) \bmod 11 \\
= & \{m = 10 \times n + d\} \\
& m \bmod 11
\end{aligned}$$

Reasoning About Programs

4.1 Overview

The program verification problem Design of any engineering artifact, say a bridge, requires a specification of its properties first, followed by a verification of the compliance of the design with the specification. The rigor of specification and verification varies widely among the various disciplines. The specification may be described using words, pictures and mathematical formulae, and verification is often carried out empirically, perhaps on a model of the real product. The underlying mathematical and physical theories from which the specification notation and design principles are derived (e.g., structural design principles for bridge design) constitute a major part of the study in the associated discipline.

Verification and analysis are especially important for computer programs because they tend to be some of the most complex artifacts designed by humans. A car manufacturer demands assurance about the quality of the software controlling its cars and its real-time response. Authorities demand a much greater level of assurance for the software controlling a nuclear power plant and for commercial aviation because the cost of error in logic or performance is astronomical. The software controlling the Mars rover Curiosity, for instance, must never fail¹ because there are very few options for human controllers on earth to overcome a major software problem. In fact, our society has come to rely on computer programs to such an extent that faulty software may be the most expensive aspect of any engineering design.

Testing a design is a time-honored way of gaining confidence in the eventual product. Testing is used extensively in all engineering disciplines. Physical testing

1. Actually, the software and hardware of Curiosity rover, launched in November 2011, did fail a few times, most notably on sol 200 (a “sol”, or Martian day, is about 40 minutes longer than an Earth day). Fortunately, the controllers on earth could transmit fixes to it, including commands to switch to a back-up computer. The rover is still operational as of February 20, 2022.

of a prototype is now increasingly being replaced by simulation on a computer, whereby a computer-based model of the design can be rapidly tested for a variety of criteria. For computer software, the design and the final product can be tested directly, and that is how most software is verified and analyzed today. Unfortunately, testing can never cover all cases except in a few special-purpose software (say, a small finite state controller). For most software, the number of cases to be tested is astronomical, if not infinite. So, testing, though useful in practice, is never foolproof.² For truly critical software, as for the Mars Rover, for instance, essential parts were not merely tested but mathematically verified with the help of a computer, using techniques similar to the ones described in this chapter.

Software is a mathematical object akin to a mathematical formula. It is written in a notation with exact meaning and its rules of execution on a computer are equally precise. Therefore, in theory, a computer program is amenable to a mathematical proof, that all possible executions of the program are consistent with its specification, in being correct and meeting the performance goals. Barring failure in hardware a correct program should continue to run flawlessly forever. This is in contrast to any other engineering product that is subject to physical deterioration over time.

A theory of software verification was developed in the 1960s; see the seminal papers by Floyd [1967] and Hoare [1969]. The early attempts in verification were entirely manual and they were successful only on very small programs. An influential monograph by Dahl et al. [1972] argued that better principles for software design could substantially reduce the amount of work needed for proofs.

Program analysis, or more accurately, resource usage analysis of a program, is a set of techniques to determine the performance of an algorithm or program without actually observing its behavior in practice. For many important algorithms such analysis has provided valuable insight and criteria to choose among the available algorithms for a problem. For complicated programs their behaviors are often hard to derive analytically; so, empirical observations are used to predict their behavior.

A mature engineering discipline supports construction of artifacts whose behaviors are predictable. As computer science has gained maturity, its artifacts, computer programs, have become more amenable to analysis. Analysis of their correctness and performance, which form the basis of their predictability, depends on the theories of correctness and complexity, which are the topics covered in this chapter.

2. A famous slogan due to Dijkstra in Dahl et al. [1972]: *program testing can be used to show the presence of bugs, but never to show their absence!*

Correctness and performance A program is taken to be correct if it satisfies its logical specification no matter how slowly it runs in practice. For example, a chess playing program that runs through all possible sequence of moves by both players to choose its next move (assuming there is a fixed limit on the number of moves in a game) can be correctly implemented though it is of little value in actually playing chess. Such an algorithm is often called a “British Museum algorithm”.

Analysis of a program involves both its correctness and its usage of resources: its running time, space usage, amount of communication with other machines or the response time for interactions with its environment. This book considers only sequential programs, so the running time and space are the only resources of interest. Henceforth, I use the generic term *cost* for the amount of resource usage.

Structure of the chapter This chapter presents both a theory of verification of sequential imperative programs³ and a theory of program analysis. A small set of the essential features of a programming language is introduced and the verification theory, which is quite small in size, is explained. Realistic programming languages include many features for modular program construction including mechanisms for defining objects. Their treatment requires a more elaborate theory. A few small examples are treated in this chapter to illustrate the main ideas of verification and program analysis. Some longer examples are analyzed in Chapter 5. The ideas developed in this chapter are used heavily throughout the rest of the book.

Font and color usage I use certain stylistic conventions in writing programs. The keywords are shown in boldface as in **if** and **then**. The commands and predicates of a program are shown in blue color, as in $x := x + 1$ or $x > 10$. Other material, such as comments and additional predicates that are needed only for the proof, are shown in normal font though they are enclosed in bold curly braces, as in $\{x \text{ is now greater than } y\}$. The font and color schemes make it easier on the eye to distinguish between the various components in the program code. Unfortunately, the colors can only be appreciated on a full color display or in a manuscript printed in color.

4.2 Fundamental Ideas

This section informally introduces the two most fundamental ideas underlying verification: *invariant* and *variant* function. An invariant is a predicate that holds whenever the program control is at a specific point in its text. A variant function is a

3. In this book I do not consider concurrent programs, my own research area of interest, in which several programs may interact during their executions. I treat correctness of functional programs in Chapter 7.

function on program states whose value decreases during the program execution, and it cannot decrease forever, so the execution is bound to terminate. I explain these ideas using a few small examples in this section, and more formally in the rest of the chapter.

Making use of puzzles I employ puzzles to introduce the fundamental ideas. Use of puzzles may seem frivolous for such an important task. The intention is not to trivialize the idea but to highlight its essence concisely, without stifling it with a long description.

4.2.1 Invariant

A program's state changes with each step during its execution. How does one describe the evolving states succinctly? One powerful mechanism, called an *invariant*, is a predicate on program states that describes what does *not* change, that is, *an invariant continues to hold no matter how the state is changed*. I introduce invariants using two small puzzles. In each case, a game starts in some initial state and a step is applied repeatedly that changes the state until the game reaches a terminating state. We can describe the evolving states of each game succinctly using invariants.

4.2.1.1 The Coffee Can Problem

The coffee can problem⁴ appears in Gries [1981, in chap. 13, p. 165]. He states the problem as follows:

A coffee can contains some black beans and white beans. The following process is repeated as long as possible:

Randomly select two beans from the can. If they have the same color, throw them out, but put another black bean in. (Enough extra black beans are available to do this.) If they are of different colors, place the white one back in the can and throw the black one away.

Execution of this process reduces the number of beans in the can by one. Repetition of this process must terminate with exactly one bean in the can, for then two beans cannot be selected. The question is: what, if anything, can be said about the color of the final bean based on the number of white beans and the number of black beans initially in the can?

It seems impossible to predict the color of the final bean because the choice of beans in each step is random. Observe, however, that each step reduces the number of white beans by either 0 or 2. Therefore, the number of white beans remains

4. Gries attributes the problem to Carel Scholten and one of the first solutions to Edsger W. Dijkstra.

even/odd if the initial number of white beans is even/odd. Since the game terminates when only one bean is left, that is, with an odd number of beans, we can assert that the final bean is white iff we start with an odd number of white beans.

The invariant, that the number of white beans remains even/odd if the initial number of white beans is even/odd, can be written as the following predicate where w is the number of white beans at any point during the game and w_0 is its initial value: $w \equiv w_0 \pmod{2}$. To prove that this predicate holds after any number of steps of the game, apply induction on the number of steps. The base case: after 0 steps $w = w_0$, so $w \equiv w_0 \pmod{2}$. The inductive hypothesis: assuming that $w \equiv w_0 \pmod{2}$ holds after i steps, for any non-negative i , show that it holds after $i + 1$ steps; that is, if the invariant holds before any step it holds afterwards as well, a fact that I have already shown informally. Then, applying induction, the predicate holds after any number of steps. The appeal to induction seems like overkill. In fact, we will not appeal to induction explicitly in proving invariants though it underlies the proof of every invariant.

4.2.1.2 Chameleons

The following problem, involving multicolored Chameleons, appears in the May 2009 issue of the *Communications of the ACM*. Given below is a colorless description that captures the essentials.

This is a one-person game that starts with three piles of chips. A step of the game consists of removing one chip each from two different piles and adding both chips to the third pile. The game terminates (reaches a final state) when no more steps can be taken, that is, there are at least two empty piles. Devise a strategy to arrive at a final state, or prove that no such strategy exists for a given initial state.

First, note that different playing strategies may lead to non-termination and termination from the same initial state. To see this, let the piles be P , Q and R , and the number of chips be, respectively, p , q , and r at any point during game. Now consider the case where each of p , q , and r are at least 2 initially. Then the following three steps restore the piles to their original state: move one chip each (1) from P and Q to R , (2) from Q and R to P , and (3) from R and P to Q . These three steps can be repeated forever. A different strategy, say with $p, q, r = 2, 2, 2$, is to move chips from P and Q to R twice to terminate the game. You can convince yourself that for some initial configurations, such as $p, q, r = 1, 2, 3$, no strategy can lead to termination. We have to find the condition on the initial configuration such that there is a strategy that will lead to termination. For the moment, I just show the condition that guarantees non-termination, that is, when there is no sequence of moves to put all chips on a single pile.

Consider the step that removes chips from P and Q and adds them to R ; it decreases both p and q by 1 each, so $p - q$ is unchanged. And both $r - p$ and $r - q$ are increased by 3 because r increases by 2 and each of p and q decreases by 1. Thus, the difference modulo 3 between any two piles is unchanged by a step. Formally, the invariant is $(p - q \equiv p_0 - q_0) \wedge (q - r \equiv q_0 - r_0) \wedge (r - p \equiv r_0 - p_0)$, where p_0, q_0 and r_0 are the initial number of chips in the corresponding piles.

The game terminates only when there are at least two empty piles, say, $p = 0$ and $q = 0$; so, $p - q \equiv 0$. From the invariant, $p_0 - q_0 \equiv 0$. That is, termination is possible only if some pair of piles are congruent modulo 3 initially. I have not shown that this condition is sufficient, that is, that given this condition there is a strategy that leads to termination. This is indeed true, and I defer the proof to Section 4.7.2 (page 173).

4.2.2 Termination

An invariant says what does not change about the program state at some program point. Invariants alone are not sufficient to prove a program's correctness. Consider a program that repeatedly executes a step that does not change the program state. Then the step preserves every predicate that holds initially but does not achieve any goal state that differs from the initial state. To prove termination, that is, that the program reaches a state from which no more steps can be taken, we need to show that some measure of the program, called a *metric*, decreases during program execution and that it cannot decrease forever. The metric, also called a *variant function*, is a function from the program states to a well-founded set (see Section 3.5), so the metric value cannot decrease forever along the well-founded order. In a large number of cases, the metric is a natural number whose value decreases in each step of program execution.

For the coffee can problem, the number of beans in the can decreases by 1 in each step and the game terminates when a single bean is left, so if we start with 20 beans, say, the game is over after 19 steps. Here, the metric is the number of beans in the can at any point in the game, and the proof of termination is trivial. Now, modify the game as follows: if the two selected beans have different colors, place the white one back in the can and throw the black one away, but if they have the same color, throw both of them out and add *two* black beans to the can. The invariant $w \equiv w_0 \pmod{2}$ still holds, but the game may not terminate, that is, there is no suitable metric.

Proof of termination, in general, can be very difficult. The “Three-halves conjecture” introduced in Exercise 23 of Chapter 3 (page 132) is a famous open problem in mathematics that asks whether a certain easy-to-state iterative process terminates.

Termination of a computer program is usually more tractable because it has been designed by humans who (may) have reason to believe that it terminates.

Partial correctness, Total correctness Invariants and variants are treated differently in the proof theory. *Partial correctness* of a program refers to the correctness of a program without regard to termination, whereas *total correctness* additionally includes termination. I develop the theory of partial correctness of imperative programs in Section 4.3 and termination in Section 4.7 (page 171).

The essential role of induction Our two small examples illustrate that induction is the essential tool for the proofs. For partial correctness, I showed that the invariant remains true after a step if it was true before the step and then applied induction to conclude that if the invariant is true initially, it is true after any number of steps. For termination, I showed that each step decreases the value of the metric, and then applied induction to conclude that the metric value will decrease indefinitely if the program does not terminate; by choosing a well-founded metric, we establish eventual termination.

4.3 Formal Treatment of Partial Correctness

Each puzzle in the previous section has the form of a very simple program, an initialization followed by a loop. For instance, for the coffee can problem the initialization sets the number of black and white beans to arbitrary non-negative integer values so that there is at least one bean in the can. And the loop repeatedly applies the step that (1) draws two beans, (2) tests the colors of the beans, and (3) depending on their colors modifies the contents of the can. The loop terminates when there is exactly one bean in the can. For such a simple program it was sufficient to postulate a single invariant and a variant function for termination. Programs, in general, are much more elaborate. So, they require multiple invariants and variant functions.

4.3.1 Specification of Partial Correctness

The starting point in proving a program is to specify its inputs and outputs. The specifications use predicates over (some of) the program variables; so, a predicate is a boolean-valued function over the program states.⁵

The *input specification* is a predicate that describes the properties of the expected input. The *output specification* is a predicate that is required to hold for the program state at its termination. Consider program *SQRT* that returns the positive square root of variable x in variable y . Its input specification may be $x \geq 0$

5. We typically see specifications written in English. Natural languages are too imprecise for control of a precise device, such as a computer.

and the output specification $y = |\sqrt{x}|$. Conventionally, these specifications are written as

$$\{x \geq 0\} \text{ *SQRT* } \{y = |\sqrt{x}|\}$$

where the input specification precedes and the output specification follows the program name, and both predicates are enclosed within braces (the braces are written in bold font in this book). The text of program *SQRT* may be placed between the input and output specifications instead of just the program name.

The meaning of the given specification is: if input x of *SQRT* satisfies $x \geq 0$, then the output y satisfies $y = |\sqrt{x}|$ provided the program execution terminates. Note that we could have written the output specification equivalently as: $y \geq 0 \wedge y^2 = x$.

The types of x and y may also form part of the specification or may be left implicit. The specification may include additional constraints on the behavior of the program, such as that the initial and final values of x are identical; see Section 4.3.3 for how to express this constraint as part of the specification.

Specifications of numerical algorithms are especially involved. When calculating with real numbers we have an obligation to be more precise about the meaning of equality, as in $y = |\sqrt{x}|$. Square root of 2, for example, is an irrational number, (see Section 2.8.1.1, page 59), so it has an infinite fractional part. A program will compute y to a finite degree of precision. So, we have to specify that y is computed to some, but not infinite, precision. In this case we may write $|y - \sqrt{x}| \leq 0.0005$, say, as the output specification.

4.3.2 Hoare-triple or Contract

The triple of input, output specifications and the program text, as shown above, is called a *Hoare-triple* in honor of its inventor Tony Hoare [1969]. I also use the term *contract* for Hoare-triple. The input specification is also called the *precondition* of the program and the output specification its *postcondition*. More generally, an *assertion* is a predicate that holds at some point in the program text, so that whenever the program control is at the given point the program state satisfies the assertion. So, both precondition and postcondition are assertions.

A *program state*, or just *state*, is an assignment of values to the program variables. A p -state is a state that satisfies predicate p . The contract $\{p\}f\{q\}$ asserts that the execution of f in any p -state results in a q -state whenever the execution of f terminates. I say that f “meets its specification” or f is “partially correct with respect to the given specification” if the contract can be established.

If f does not terminate for some initial p -state, it is partially correct by default for that input. If f never terminates for any input, then it is partially correct with

respect to every contract. This is a startling departure from the common usage of the term “correct”.

Predicate *true* is sometimes used as a precondition to denote that the program may start execution in any state (obeying only the implicit constraints, if any, about the types of variables). And, *true* holds, trivially, as a postcondition for any program. A pathological case is the precondition *false*; then any postcondition holds.

- Example 4.1** 1. Given a finite set of items S , program *LOOKUP* sets boolean variable *found* to *true* iff a given value v is in S .

$$\{true\} \text{ LOOKUP } \{found \equiv (v \in S)\}.$$

The specification implicitly assumes that S is not modified. To state this explicitly, we need auxiliary variables (see Section 4.3.3).

2. Program *Sort3* permutes the values of integer-valued variables p , q and r that are initially distinct such that they are sorted in increasing order if and when the program terminates.

$$\{p \neq q, q \neq r, r \neq p\} \text{ Sort3 } \{p < q < r\}.$$

The specification almost, but does not quite, capture our intent. A program that assigns any three distinct values to p , q and r satisfies the specification. We need to say that the values of p , q and r are drawn from their initial values, for which we need to use auxiliary variables (see Section 4.3.3).

4.3.3 Auxiliary and Derived Variables

Auxiliary variable An *auxiliary variable* does not occur in a program but is introduced to make reasoning about it simpler. Auxiliary variables are introduced into a program in such a manner that they neither alter the program execution for any input nor affect the values of the program variables. Therefore, the commands of the program are not changed, but assignments of the form $x := e$, where x is an auxiliary variable, may be added to the program. Here e may name x , other auxiliary variables and program variables. Typically, auxiliary variables record a history of the computation. The term *logical variable* is often used to denote an auxiliary variable whose value does not change. A logical variable z is typically used to record the value of a program variable x ; later a contract relates a new value of x to its old value in z .

A common use of auxiliary variables is to record the initial values of program variables so that the postcondition may state a property relating the initial and final values of the variables. Consider the specification of *SQRT* given in Section 4.3.1

$$\{x \geq 0\} \text{ SQRT } \{y = \sqrt{x}\}.$$

Now add the requirement that the initial and final values of x are identical. Let x_0 be an auxiliary variable that is not named in *SQRT*; we use it in the precondition to record the initial value of x .

$$\{x = x_0, x \geq 0\} \text{SQRT} \{x = x_0, y = +\sqrt{x}\}.$$

If *SQRT* does modify x but computes the positive square root of the initial value of x in y , the following specification suffices.

$$\{x = x_0, x \geq 0\} \text{SQRT} \{y = +\sqrt{x_0}\}.$$

Example 4.2 I write more complete contracts for the ones shown in Example 4.1.

1. S_0 and v_0 are auxiliary variables.
 $\{S = S_0, v = v_0\} \text{LOOKUP} \{found \equiv (v_0 \in S_0)\}.$
2. p_0, q_0 and r_0 are auxiliary variables.
 $\{p, q, r = p_0, q_0, r_0, p \neq q, q \neq r, r \neq p\}$
 Sort3
 $\{ \{p, q, r\} = \{p_0, q_0, r_0\}, p < q < r \}.$

Derived variable A *derived variable* is a program variable that is a function of program variables and, possibly, other derived variables. For example, a program may have variables *males*, *females* and *persons* for the number of males, females and the total number of males and females. Program description would be simpler if we identify *persons* as a derived variable, and declare it:

$$\text{derived } persons = males + females$$

Then we never assign any value to *persons*, implicitly delegating its computation after execution of every command to the computer. There is no restriction on the usage of a derived variable in a program. We can assume that the definition, $persons = males + females$, is an invariant, that it holds at all times during program execution without having to prove it.

One has to be careful in estimating the running time of a program in the presence of derived variables because the running time must include the time to compute the derived variable values at each step. If function computation for each derived variable takes unit time (which I will later write as $\mathcal{O}(1)$), the extra computation time is proportional to the number of execution steps. Typically, derived variables are used in abstract programs and discarded or replaced by other variables in concrete programs.

4.4 A Modest Programming Language

I illustrate the proof theory using a modest programming language. The proof theory can be extended to more realistic programming languages. I assume that

variables are declared in the accompanying text, so I do not show the declarations in the program code.

The programming language is so modest that there are no input and output commands. This is unrealistic in any practical programming language. However, it is no impediment to explaining the algorithms I present in the book, and it keeps the proof theory simpler.

A program is a *command*. A command is defined (recursively) as shown next.

1. **Simple command:** The simple command *skip* does nothing; it always terminates without altering the program state. A *simple assignment* command $x := e$, where x is a variable and e an expression, is also a simple command. This command is executed by first evaluating e and then assigning its value to x . The command does not terminate if the evaluation of e does not terminate.

Multiple assignment Multiple assignment is a useful generalization of simple assignment. It is of the form $x, y, z := e, f, g$, where x, y and z are distinct variables and e, f and g are expressions. The expressions in the right side, e, f and g , are first evaluated and their values are then assigned to the variables x, y and z . For example, $x, y := y, x$ exchanges the values of x and y . And, given that m and n are two consecutive Fibonacci numbers, $m, n := n, m + n$ computes the next pair of consecutive Fibonacci numbers.

Multiple assignment is introduced only for convenience in programming. It can always be replaced by a sequence of simple assignments.

2. **Sequencing:** Given commands f and g , $f; g$ is a command whose execution in a given program state proceeds by executing f then executing g (g starts on termination of f). If either f or g does not terminate, then neither does $f; g$.
3. **Conditional:** Given predicate b and commands f and g

if b then f else g endif

is a command that is executed in a given program state according to the following rules. If predicate b is *true* in the current program state, then f is executed, and if it is *false*, then g is executed. If the evaluation of b does not terminate, or the value of b is *true* and the execution of f does not terminate, or the value of b is *false* and the execution of g does not terminate, then the execution of the conditional does not terminate.

4. **Loop:** There are two forms of loop commands used in this book: **while** and **for**. The **for** command is not strictly necessary because it can be replaced by a **while** command with slightly extra effort; I include it because it makes some program codes much simpler.

The **while** command has the form, for predicate b and command f :

while b do f enddo

This command is executed by first evaluating predicate b . If b 's value is *false*, the execution terminates, if b is *true*, f is executed as an iteration of the loop body, and following the completion of the iteration, the execution of the loop is repeated in the resulting state. If the evaluation of b or the execution of f does not terminate in some iteration, then the execution of the loop does not terminate. Further, even if each iteration terminates, the loop execution may not terminate because of a never-ending sequence of iterations.

The **for** command has the form, for a finite set or sequence S and variable x :

for $x \in S$ **do** f **endfor**

The command executes f for each value of x in S , in arbitrary order if S is a set and in the prescribed order for a sequence. Set or sequence S is specified using the notations described in Chapter 2. Any sequence S used in this context has distinct items. I assume that S , set or sequence, is not modified by the execution of f .

The following program computes the sum of the values in an array $A[0..N]$ in variable sum .

$sum := 0; i := 0;$
while $i < N$ **do** $sum := sum + A[i]$ **enddo**

Contrast the solution with a program using **for**.

$sum := 0;$
for $i \in 0..N$ **do** $sum := sum + A[i]$ **endfor**

Observe that **for** $i \in 0..N$ **do** can be replaced by **for** $i \in \{0..N\}$ **do**, that is, the elements of A can be added to sum in arbitrary order.

5. **Non-deterministic assignment:** This command, written as $x : \in S$, assigns *any* element of set S to x . For example, $x : \in \{i \mid 0 \leq i < 10\}$ assigns any value in the set $\{0..10\}$ to x . I permit only a finite set S . In case S is empty, the execution of the command does not terminate.⁶

The following program processes all elements of set S in arbitrary order.

while $S \neq \{\}$ **do**
 $x : \in S$; “process x ”; $S := S - \{x\}$
enddo

Non-determinism is an essential ingredient in concurrent programming though not in sequential programming. So, non-deterministic assignment

⁶ Dijkstra is credited with introducing nondeterminism in programming in [Dijkstra \[1975\]](#). The notation $: \in$ has been used by Eric C. R. Hehner as early as 1984 in [Hehner \[1984\]](#).

is not strictly necessary for this book. I introduce this command solely to simplify descriptions of certain algorithms by raising the level of abstraction. At some point in a design of an algorithm, it is immaterial which specific value is assigned to a variable. It is, in fact, advantageous to underspecify because it presents a variety of options for later implementation.

I use this command to assign any value to x from the non-empty set $\{y \mid b(y)\}$. For example, to assign any Pythagorean triple to integer variables x, y and z , where z is to be smaller than 100, use

$$(x, y, z) : \in \{(x, y, z) \mid x^2 + y^2 = z^2, 0 < z < 100\}.$$

Multiple assignment command can be generalized with non-deterministic assignment, as in $x, y : \in S, T$. Here x and y are distinct variables, and S and T are sets. The command execution terminates only if both S and T are non-empty; then x is assigned any value in S and y in T . Command $x, y : \in S, S$ may assign identical values to x and y ; it is of dubious value in practice.

6. **Procedure call:** A procedure declaration is of the form

Proc $P(x)$ S **endProc**

where P is the name of the procedure, x a list of its *formal parameters* and S its body. The formal parameter list may be empty; then the declaration is of the form $P()$. A call to this procedure, $P(a)$, can be made only if P is in the scope of the call.⁷ Execution of the call $P(a)$ first assigns the values of the *actual parameters* a to the corresponding formal parameters x . Then the body S of P is executed. On termination of P the values computed in the formal parameters are assigned to the corresponding actual parameters. The non-local variables in the scope of P may also be accessed and modified during its execution.⁸ Execution of a procedure may not terminate. In this chapter, I assume that functions and procedures are *not* parameters of a procedure; see Section 7.2.9 (page 400) for treatment of higher-order functions that allow for this possibility.

Procedure P may call itself (recursively) within its body S . More generally, a set of procedures may call each other in a mutually recursive fashion. I show examples of mutual recursion in Chapter 7.

7. I do not explain the concept of “scope”, assuming that the reader is familiar with this basic concept of imperative programming.

8. Many programming languages distinguish between *value* and *variable* parameters. Value parameters are expressions and their values do not change during the execution of the procedure. Variable parameters are variable names and only these variables may be assigned values. I have eliminated the distinction here for simplicity in the proof theory.

7. **Function call:** A function declaration is of the form

Funct $f(x)$ S **endFunct**

where f is the name of the function, x a list of its *formal parameters* and S its body. Typically, the parameter list is non-empty, otherwise the function value is constant. Functions and procedures are alike in most respects. I outline the differences here.

Parameter values of a function can only be read but not modified within the function body. For example, for function $f(x, y)$ with formal parameters x and y , calling $f(a, b)$ first assigns the values of actual parameters a and b to x and y , respectively, executes the body of f , and then returns the result as $f(a, b)$. A function call is an expression, so it may be embedded in other expressions, as in $plus(times(a, b), times(c, d))$, where $plus$ and $times$ are functions. All functions that we consider are side-effect free. The execution of a function call, like a procedure call, may not terminate.

A function may be recursive, so that f is called within its body S .

Our simple (and simplistic) treatment of programs omits important features such as declarations of variable types and data objects. Yet this minimal language is sufficient to illustrate the essential ideas of program proving.

4.5 Proof Rules

A *proof rule* permits deriving one or more contracts from a given set of contracts and propositions. The given set of contracts and propositions are called assumptions/antecedents/hypotheses and the derived contracts are consequents/conclusions. In the written form, hypotheses are written on top of a horizontal bar and the conclusions below the bar. A proof rule asserts that if all the hypotheses above the bar can be proved, then the contracts below the bar hold. A proof rule that has no hypothesis is written without the horizontal bar. A sample proof rule looks like:

$$\frac{\{p\} f \{r\}, \{r\} g \{q\}}{\{p\} f; g \{q\}}$$

Observe that neither a contract nor a proof rule is a predicate, so the proof rule cannot be written as just an implication connecting the (conjunction of the) hypotheses to the consequences.

There is one proof rule for each programming construct in our simple language. Proof of a program proceeds by starting with its simple commands and proving their associated contracts using the proof rules. Next, the contracts for larger components are derived from the hypotheses of their constituent components using the appropriate proof rules.

In the proof rules below, p , q and r are predicates over arbitrary variables, and f and g are commands.

4.5.1 Rule of Consequence

Suppose $\{p\} f \{q\}$ holds and predicate p' satisfies $p' \Rightarrow p$. Then we can claim that $\{p'\} f \{q\}$ holds as well. This is because the p' -state before execution of f can be regarded as a p -state. Analogously, given $\{p\} f \{q\}$ and $q \Rightarrow q'$, assert $\{p\} f \{q'\}$. Formally:

$$\frac{p' \Rightarrow p, \{p\} f \{q\}}{\{p'\} f \{q\}} \qquad \frac{\{p\} f \{q\}, q \Rightarrow q'}{\{p\} f \{q'\}}$$

Combining both rules (use p' in place of p in applying the second rule), we have:

$$\frac{p' \Rightarrow p, \{p\} f \{q\}, q \Rightarrow q'}{\{p'\} f \{q'\}}$$

4.5.2 Simple Commands

skip command The *skip* command does not change the program state. So, its precondition continues to hold on termination of its execution. Therefore, any assertion that follows from the precondition also holds. We codify this discussion by the following proof rule:

$$\frac{p \Rightarrow q}{\{p\} \text{skip} \{q\}}$$

For example, we can claim $\{x > y\} \text{skip} \{x + 1 > y\}$.

Assignment command

“Life can only be understood backwards; but it must be lived forwards”.

— Søren Kierkegaard, Danish Philosopher

The proof rule for an assignment command, say $x := y$, may seem obvious: if p is the precondition of the command, then its postcondition is obtained by substituting x for every occurrence of y in p . Therefore, we can assert that $\{y > 0\} x := y \{x > 0\}$. Unfortunately, this simple scheme does not work for $x := e$ where e is an expression, not a simple variable. For example, it is hard to derive q in $\{x > 0\} x := x^2 - 2 \times x \{q\}$.

As Kierkegaard tells us, even though the execution of the assignment command moves forward, to understand it we need to work backward. Given an assignment command $x := e$ and a *postcondition* q , we can determine its precondition. For example, if $x > 0$ holds after the execution of $x := e$, then $e > 0$ has to hold before its execution because x acquires the value of e . Adopt the notation $q[x := e]$ for the predicate obtained by replacing every occurrence of x in q by e . Then, q holds after

the execution of $x := e$ if and only if predicate $q[x := e]$ holds prior to its execution. So, the proof rule for the assignment is:

$$\{q[x := e]\} x := e \{q\}$$

This is often called the *axiom of assignment*, or the *backward substitution* rule, for obvious reasons.

As examples, the following contracts are satisfied:

$$\begin{array}{lll} \{x + 1 > 5\} & x := x + 1 & \{x > 5\} \\ \{y + z > 5\} & x := y + z & \{x > 5\} \\ \{x^2 - 2x > 0\} & x := x^2 - 2 \times x & \{x > 0\} \\ \{A[i - 1] > A[j]\} & i := i - 1 & \{A[i] > A[j]\} \end{array}$$

The backward substitution rule assumes that execution of $x := e$ does not affect the value of any variable mentioned in e other than x . That is, the variables are not aliased, so, x and y are not the same variable in, say, $x := y + z$. Equivalently, assignment has no side effect.

Assignment to array items The rules for assignment to array items, and to parts of any structured object such as a tuple, follow a slightly different rule. Consider the assignment $A[i] := A[j] + 1$ with the postcondition $A[i] > A[j]$. The application of the backward substitution rule yields $A[j] + 1 > A[j]$, that is, *true* as the precondition. But this is wrong; if $i = j$ prior to the assignment, $A[i] > A[j]$ can not hold as a postcondition.

We regard assignment to an array item as an assignment to the whole array, that is, $A[i] := A[j] + 1$ is the assignment $A := A'$ where A' is an array defined below. After the assignment $A[i] := A[j] + 1$, we expect all items of A other than $A[i]$ to remain the same, and $A[i]$ to acquire the value $A[j] + 1$. So,

$$A'[k] = \begin{cases} A[k] & \text{if } k \neq i \\ A[j] + 1 & \text{if } k = i \end{cases}$$

Now apply the backward substitution rule to the assignment $A := A'$ with the postcondition $A[i] > A[j]$. This yields the precondition $A'[i] > A'[j]$ from which deduce $i \neq j$.

I use the notation $A[i := A[j] + 1]$ for A' ; this is a slight overloading of the notation for substituting an expression for a variable in a predicate. So, $A[i := A[j] + 1][k] = A[k]$ if $k \neq i$, and $A[j] + 1$ if $k = i$. Then the assignment $A[i] := A[j] + 1$, which is $A := A'$, is equivalent to $A := A[i := A[j] + 1]$.

Multiple assignment The proof rule for multiple assignment is similar to that for single assignment:

$$\{q[x, y, z := e, f, g]\} x, y, z := e, f, g \{q\}$$

where $q[x, y, z := e, f, g]$ is the predicate obtained by replacing every occurrence of x, y and z in q by e, f and g *simultaneously*. For example,

$$\{y > x + 1\} \text{ } x, y := y, x \text{ } \{x > y + 1\}, \text{ and } \\ \{n, m + n = \text{Fib}(i + 1), \text{Fib}(i + 2)\} \text{ } m, n := n, m + n \text{ } \{m, n = \text{Fib}(i + 1), \text{Fib}(i + 2)\}$$

Multiple assignment with array variables should be treated with care. An assignment like $A[i], A[j] := 3, 5$ requires the precondition $i \neq j$.

4.5.3 Sequencing Command

To prove $\{p\}f; g\{q\}$, I introduce an intermediate assertion r that holds right after the execution of f and before g . If the contracts $\{p\}f\{r\}$ and $\{r\}g\{q\}$ can be established, then surely $\{p\}f; g\{q\}$ can be established because the execution of f in a p -state can only terminate in a r -state, and the execution of g in a r -state can only terminate in a q -state. Formally:

$$\frac{\{p\}f\{r\}, \{r\}g\{q\}}{\{p\}f; g\{q\}}$$

The proof rule for a sequence of commands is analogous:

$$\frac{\{p_0\}f_0\{p_1\}, \dots, \{p_i\}f_i\{p_{i+1}\}, \dots, \{p_n\}f_n\{p_{n+1}\}}{\{p_0\}f_0; \dots f_i; \dots f_n\{p_{n+1}\}}$$

The intermediate assertions can be systematically computed if the commands are all assignment commands by applying the backward substitution rule.

Example 4.3 I prove the correctness of a simple program, one that exchanges the values of variables x and y without using a temporary variable:

$$x := x + y; y := x - y; x := x - y.$$

Show that if the initial values of x and y are X and Y , respectively, then their eventual values are Y and X . That is, I claim:

$$\{x, y = X, Y\} x := x + y; y := x - y; x := x - y \{x, y = Y, X\}.$$

Applying the backward substitution rule to the last assignment command, $x := x - y$, I compute its precondition as $(x - y, y = Y, X)$. This serves as the postcondition of the second assignment, $y := x - y$. Its precondition is similarly computed to be $(x - (x - y), x - y = Y, X)$, or $(y, x - y = Y, X)$. Apply a similar computation to the first assignment, $x := x + y$, to get the initial precondition $(y, x + y - y = Y, X)$, that is, $(y, x = Y, X)$. So, we have shown that

$$\{y, x = Y, X\} x := x + y; y := x - y; x := x - y \{x, y = Y, X\} \text{ holds.}$$

Nitpicking It is now tempting to assert that we have established our goal, but that is not quite correct because the specification had $(x, y = X, Y)$ as the initial precondition whereas we have established $(y, x = Y, X)$ as the initial precondition. This is

a minor point, but to be precise we apply one more rule, the rule of consequence: from $(x, y = X, Y) \Rightarrow (y, x = Y, X)$ and the established claim, we can conclude

$$\{x, y = X, Y\} \ x := x + y; \ y := x - y; \ x := x - y \ \{x, y = Y, X\}.$$

I will not be so pedantic in our proofs in the rest of this book. But, it is essential that any omitted step be defensible.

Example 4.4 This example illustrates how designs of small program segments can be guided by the proof theory. Determine expression e so that the following contract is met:

$$\{sum = (+j : 0 \leq j < n : A[j])\} \ sum := e \ \{sum = (+j : 0 \leq j \leq n : A[j])\}.$$

Using backward substitution, $e = (+j : 0 \leq j \leq n : A[j])$ is a precondition, so we have $(sum = (+j : 0 \leq j < n : A[j])) \Rightarrow (e = (+j : 0 \leq j \leq n : A[j]))$. Rewriting, $(sum = (+j : 0 \leq j < n : A[j])) \Rightarrow (e = sum + A[n])$. Therefore, we let e be $sum + A[n]$; so the assignment is $sum := sum + A[n]$.

An observation about command skip Intuitively, we know that f is the same as $f; skip$ or $skip; f$. This intuition can be justified as follows. I show that whatever contract can be proved about f can be proved about $f; skip$ and conversely (proofs about $skip; f$ are analogous).

Suppose $\{p\}f\{q\}$ holds. Then $\{p\}f; skip\{q\}$ holds by applying the sequencing rule to: (1) $\{p\}f\{q\}$ and (2) $\{q\}skip\{q\}$.

Conversely, suppose $\{p\}f; skip\{q\}$ holds; I show that $\{p\}f\{q\}$ holds. In order to establish $\{p\}f; skip\{q\}$ using the sequencing rule, the contracts $\{p\}f\{r\}$ and $\{r\}skip\{q\}$ hold, for some assertion r . From the proof rule of *skip*, deduce $r \Rightarrow q$. Now, using $\{p\}f\{r\}$ and $r \Rightarrow q$ together with the rule of consequence conclude $\{p\}f\{q\}$.

4.5.4 Conditional Command

To construct the proof rule for $\{p\} \text{ if } b \text{ then } f \text{ else } g \text{ endif } \{q\}$, argue as follows. Suppose the conditional is executed in a p -state when b also holds; then f is executed under the precondition $p \wedge b$ and it must establish q as a postcondition. That is, $\{p \wedge b\}f\{q\}$. Analogously, if $\neg b$ holds in the state when the conditional is executed, then g is executed under the precondition $p \wedge \neg b$ and it must establish q as a postcondition. That is, $\{p \wedge \neg b\}g\{q\}$. Combining these observations, we have the proof rule for the conditional command:

$$\frac{\{p \wedge b\}f\{q\}, \{p \wedge \neg b\}g\{q\}}{\{p\} \text{ if } b \text{ then } f \text{ else } g \text{ endif } \{q\}}$$

Example 4.5 The following program segment computes the absolute value of a real-valued variable x in y : **if** $x \geq 0$ **then** $y := x$ **else** $y := -x$ **endif**. To prove its partial correctness, we have to prove the following contract.

$$\{x = X\} \text{ if } x \geq 0 \text{ then } y := x \text{ else } y := -x \text{ endif } \{y \geq 0 \wedge (y = X \vee y = -X)\}$$

which requires proving the following two contracts:

$$\{x = X \wedge x \geq 0\} y := x \{y \geq 0 \wedge (y = X \vee y = -X)\} \text{ and,} \\ \{x = X \wedge x < 0\} y := -x \{y \geq 0 \wedge (y = X \vee y = -X)\}$$

Prove both using the backward substitution rule. Next, prove the stronger result that x is not modified by the program.

A variation of the conditional command A special case of the conditional is: **if** b **then** f **endif**. This is equivalent to: **if** b **then** f **else skip** **endif**. So its proof rule is easily obtained from the general case:

$$\frac{\{p \wedge b\} f \{q\}, p \wedge \neg b \Rightarrow q}{\{p\} \text{ if } b \text{ then } f \text{ endif } \{q\}}$$

4.5.5 Loop Command

Much of programming and proof theory would be unnecessary without some form of repetition (or recursion) in a programming language. Proofs of programs without loops are entirely straightforward because all the required intermediate assertions can be systematically generated. Much of the proof can be relegated to a computer in that case.

Proof of a loop command requires postulating an invariant, also called a *loop invariant*. As we have seen in Section 4.2.1, this requires a deep understanding of the loop computation, a task at which computers are not yet proficient.

Proof rule: *while* loop

Consider a loop of the form **while** b **do** f **enddo**. Define p to be a *loop invariant* if: an execution of f starting in a p -state terminates, the resulting state is also a p -state. Then, using elementary induction, if the loop is started in a p -state, it establishes a p -state whenever it terminates. Observe that an iteration is started only if the loop iteration condition, b , holds, and when the loop terminates the loop exit condition, $\neg b$, holds. So, we have the following proof rule for the loop command:

$$\frac{\{p \wedge b\} f \{p\}}{\{p\} \text{ while } b \text{ do } f \text{ enddo } \{p \wedge \neg b\}}$$

Example 4.6 Consider the **Simple multiplication** program in Figure 4.1. It multiplies integers x and y and stores the result in z . Introduce auxiliary variable $prod$ to denote the initial value of $x \times y$.

The proof requires a loop invariant p that has the following properties: (1) p holds immediately after the initialization of z , that is, $\{x \times y = prod\} z := 0 \{p\}$; (2) p is a loop invariant, that is, $\{p \wedge y \neq 0\} z := z + x; y := y - 1 \{p\}$; and (3) the postcondition of the loop implies the given postcondition of the program, that is,

Simple multiplication

```

{ $x \times y = prod$ }
 $z := 0$ ;
while  $y \neq 0$  do
   $z := z + x$ ;
   $y := y - 1$ 
enddo
{ $z = prod$ }

```

Figure 4.1

$(p \wedge y = 0) \Rightarrow (z = prod)$. Therefore, p should state a relationship among x , y and z that satisfies the conditions (1), (2) and (3).

There is no recipe for devising p , but observe that every iteration increases z by x and decreases y by 1, and x is unchanged. Thus, $z + x \times y$ is unchanged. The initial value of $z + x \times y$ is $prod$, so I propose as invariant: $z + x \times y = prod$. A quick informal check shows that requirements (1) and (3) are met. Next, let us attempt a formal proof.

The overall contract is $\{x \times y = prod\} f; g \{z = prod\}$, where f is $z := 0$ and g is the given loop. I prove this contract using the sequencing rule with p as the intermediate assertion: $\{x \times y = prod\} z := 0 \{z + x \times y = prod\}; g \{z = prod\}$. The proof of $\{x \times y = prod\} z := 0 \{z + x \times y = prod\}$ is immediate using the backward substitution rule and elementary arithmetic.

Next, prove $\{z + x \times y = prod\} g \{z = prod\}$. The contract is in a form that is suitable for the application of the loop proof rule: the precondition is $z + x \times y = prod$ and $z + x \times y = prod \wedge y = 0$ implies the postcondition. Therefore, the remaining proof obligation is to show that the hypothesis of the loop proof rule is met, that is,

$$\{z + x \times y = prod \wedge y \neq 0\} z := z + x; y := y - 1 \{z + x \times y = prod\}.$$

Apply the backward substitution rule twice to establish this fact. So the proof of partial correctness is complete.

Note Observe that the given program is actually incorrect because it does not terminate for negative or non-integer values of y , which are permitted by the precondition of the program. We have proved only partial correctness, that whenever the program terminates it delivers the correct result.

Proof rule: for loop

A **for** loop has the form: **for** $x \in S$ **do** f **endfor**

where S is either a set or a sequence. The two forms of S have slightly different proof rules. In each case the goal is to establish $p(S)$ as a postcondition for a given predicate p . The proof rules embody induction on the structure of S .

- S is a set: (1) show that $p(\{\})$ holds, and (2) for any $T \cup \{x\} \subseteq S$, if $p(T)$ is a precondition of f , then $p(T \cup \{x\})$ is a postcondition. Further, f does not modify S .

$$\frac{\{p(T)\} f \{p(T \cup \{x\})\}, \text{ where } T \cup \{x\} \subseteq S, \{S = S_0\} f \{S = S_0\}}{\{p(\{\})\} \text{ for } x \in S \text{ do } f \text{ endfor } \{p(S)\}}$$

- S is a sequence: Below, $T ++ x$ is the sequence obtained by concatenating x at the end of T . Show that (1) $p(\langle \rangle)$ holds, and (2) for any prefix $T ++ x$ of S , if $p(T)$ is a precondition of f , then $p(T ++ x)$ is a postcondition. Further, f does not modify S .

$$\frac{\{p(T)\} f \{p(T ++ x)\}, \text{ where } T ++ x \text{ is a prefix of } S, \{S = S_0\} f \{S = S_0\}}{\{p(\langle \rangle)\} \text{ for } x \in S \text{ do } f \text{ endfor } \{p(S)\}}$$

Example 4.7 Consider the program shown as an example of looping constructs.

```
sum := 0;
for i ∈ 0..N do sum := sum + A[i] endfor
```

To prove the postcondition $sum = (+j : 0..N : A[j])$, postulate the invariant $sum = (+j : 0..i : A[j])$. The invariant holds after the initialization. Apply the proof rule for a sequence:

$\{sum = (+j : j \in 0..i : A[j])\} \text{ sum} := \text{sum} + A[i] \{sum = (+j : j \in 0..(i+1) : A[j])\}$.
Its proof is straightforward.

4.5.6 Non-deterministic Assignment

To motivate the proof rule for $x \in S$, consider a simple case, $x \in \{3, 5\}$. To show that q holds as a postcondition, consider the two possible assignments: (1) $x := 3$, for which $q[x := 3]$ has to be a precondition, and (2) $x := 5$, for which $q[x := 5]$ has to be a precondition. Since q is required to hold as a postcondition in either case, the required precondition for $x \in \{3, 5\}$ is $q[x := 3] \wedge q[x := 5]$. This argument applies even when the elements of S are defined using x , as in $x \in \{x, x + 1\}$. Then the precondition, corresponding to postcondition q , is $q[x := x] \wedge q[x := x + 1]$.

Extending this argument, the proof rule for the general case is:

$$\{(\forall y : y \in S : q[x := y])\} x \in S \{q\}$$

The given proof rule is valid even when S is empty. Then the proof rule is

$$\{(\forall y : y \in \{\} : q[x := y])\} x \in \{\} \{q\},$$

in which the precondition is *true*. Since the execution of the command is not guaranteed to terminate, this assertion is valid for partial correctness.

If S is defined using set comprehension, we get the rules:

$$\begin{array}{lll} \{(\forall y : p(y) : q[x := y])\} & x \in \{y \mid p(y)\} & \{q\} \\ \{(\forall y : p(y) : q[x := f(y)])\} & x \in \{f(y) \mid p(y)\} & \{q\} \end{array}$$

As an example, given $x \in \{x + i \mid 1 \leq i \leq 5\}$ and the postcondition $x \leq 3$, compute the precondition as $(\forall i : 1 \leq i \leq 5 : x + i \leq 3)$, which can be simplified to $x + 5 \leq 3$.

In many cases S itself is not modified by the execution of the command, that is, $\{S = S_0\} \ x : \in S \ \{S = S_0\}$. Then we can assert that

$$\{true\} \ x : \in S \ \{x \in S\}, \text{ and } \\ \{true\} \ x : \in \{y \mid p(y)\} \ \{p(x)\}.$$

4.5.7 Procedure Call

For a procedure declaration of the form **Proc** $P(x) \ S$ **endProc**, the specification is given over P 's formal parameters and the non-local variables, G , that P may read and/or write:

$$\{p(x, G)\} \text{Proc } P(x) \ S \text{ endProc } \{q(x, G)\}$$

A procedure call $P(a)$ has the specification $\{p(a, G)\} \ P(a) \ \{q(a, G)\}$.

Example 4.8 Procedure $\text{intlog}(m, n)$ for positive integer m returns in n the integer logarithm in base 2 of m , that is, n is the largest natural number such that 2^n does not exceed m . The procedure does not modify m . Its specification is:

$$\{m = M, M > 0\} \\ \text{Proc } \text{intlog}(m, n) \ S \text{ endProc} \\ \{m = M, 2^n \leq M < 2^{n+1}\}$$

Then, for instance, a caller of $\text{intlog}(3 \times a - b, j)$ can assert:

$$\{3 \times a - b = M, M > 0\} \ \text{intlog}(3 \times a - b, j) \ \{3 \times a - b = M, 2^j \leq M < 2^{j+1}\}.$$

And, a caller of $\text{intlog}(15, k)$ can assert, after simplification of the pre- and postconditions:

$$\{true\} \ \text{intlog}(15, k) \ \{2^k \leq 15 < 2^{k+1}\}.$$

Observe that the caller of a procedure need not know how it is implemented.

Example 4.9 1. Procedure call $\text{choose}(i, j, h)$ returns an arbitrary value of h strictly between i and j , provided $i + 1 < j$ is a precondition. It does not modify i or j .

$$\{i = i_0, j = j_0, i + 1 < j\} \\ \text{Proc } \text{choose}(i, j, h) \ S \text{ endProc} \\ \{i = i_0, j = j_0, i < h < j\}$$

Observe that different calls to $\text{choose}(0, 10, j)$, for instance, may return different values of j .

2. Procedure call $get(A, x)$ chooses an arbitrary value x from a non-empty set A . It does not modify A . As in the last example, different calls to $get(A, x)$ with the same A may return different values of x .

$$\{A = A_0, A \neq \{\}\} \text{ Proc } get(A, x) \ S \ \text{endProc} \ \{A = A_0, x \in A\}$$

3. Procedure call $fib(m, n)$, where m and n are consecutive Fibonacci numbers, say $Fib(i)$ and $Fib(i + 1)$, modifies both parameters, storing $Fib(i + 1)$ in m and $Fib(i + 2)$ in n . Below, i is any natural number.

$$\begin{aligned} & \{m = Fib(i), n = Fib(i + 1)\} \\ & \text{Proc } fib(m, n) \ S \ \text{endProc} \\ & \{m = Fib(i + 1), n = Fib(i + 2)\} \end{aligned}$$

Function specification Specification of function f with formal parameters x is a predicate of the form $q(x, f(x))$ relating the values of the formal parameters to the function value.

As an example, consider function $fintlog(m)$ that mimics the procedure $intlog((m, n))$, given earlier, where the function value is n . Its specification is given by (ignoring the types of the parameters and the function):

$$m > 0 \Rightarrow 2^{fintlog(m)} \leq m < 2^{fintlog(m)+1}$$

4.5.8 Procedure Implementation

The following proof rule for $\text{Proc } P(x) \ S \ \text{endProc}$ says that P meets its specification if S does. Below, G denotes the non-local variables of P .

$$\frac{\{p(x, G)\} \ S \ \{q(x, G)\}}{\{p(x, G)\} \ \text{Proc } P(x) \ S \ \text{endProc} \ \{q(x, G)\}}$$

A recursive call, $P(a)$, may appear within S . There is no special rule for the treatment of recursion; in proving S in the antecedent, use: $\{p(a, G)\} \ P(a) \ \{q(a, G)\}$. Proof of a recursive procedure appears in Section 5.4.

Proof rule: Function Let function f have formal parameters x , body S and specification $q(x, f(x))$. Construct procedure $F(x, y)$ with body S' , which is the same as S except that the function value is assigned to a fresh variable y . Prove:

$$\{x = x_0\} \ S' \ \{x = x_0, q(x, y)\}.$$

For example, for function $fintlog(m)$, given earlier, modify its body to S' by adding variable n , assign the function value to n , and then prove:

$$\{m = M\} \ S' \ \{m = M, m > 0 \Rightarrow 2^n \leq m < 2^{n+1}\}.$$

4.5.9 Program Annotation

The proof of **Simple multiplication** program in Figure 4.1 (page 158) is rigorous in that it follows the systematic applications of the proof rules. However, the proof structure is not evident from the given description. In order to better display the

proof structure, I adopt a convention for *program annotation* that interleaves the text of the program with the assertions required for its proof. Every piece of the program, from simple commands to nested commands, is preceded by its precondition and followed by its postcondition. A pair of assertions, p and q , that appear next to each other without a command to separate them denote that the latter follows from the former by logical implication. An annotated version of **Simple multiplication** program is given in Figure 4.2.

Annotated Simple multiplication

```

{ $x \times y = prod$ }
{ $0 + x \times y = prod$ }
   $z := 0$ ;
{ $z + x \times y = prod$ }
  while  $y \neq 0$  do
    { $z + x \times y = prod \wedge y \neq 0$ }
    { $z + x + x \times (y - 1) = prod$ }
     $z := z + x$ ;
    { $z + x \times (y - 1) = prod$ }
     $y := y - 1$ 
  { $z + x \times y = prod$ }
  enddo
{ $z + x \times y = prod \wedge y = 0$ }
{ $z = prod$ }

```

Figure 4.2

A more elaborate example The **Better multiplication** program in Figure 4.3 (page 162) employs a more sophisticated strategy for multiplying x and y . It has

Better multiplication

```

{ $x \times y = prod$ }
   $z := 0$ ;
  while  $y \neq 0$  do
    if  $odd(y)$  then  $z := z + x$  else skip endif ;
     $y := y \div 2$ ;
     $x := 2 \times x$ 
  enddo
{ $z = prod$ }

```

Figure 4.3

the same loop invariant, but instead of decreasing y by 1 it halves y in each iteration. Below, $y \div 2$ returns the (integer) quotient of dividing y by 2; $y \div 2$ can be computed by right-shifting the bit representation of y by 1 and $2 \times x$ by left-shifting x by 1. This technique reduces the number of iterations from Y to $\log_2 Y$, where Y is the initial value of y , and $Y \geq 1$; for $Y = 0$, the loop goes through zero iterations.

An iteration is based on the following two observations. If y is even, say $2 \times t$, $z + x \times y = z + x \times (2 \times t) = z + (2 \times x) \times t$. So, x may be doubled and y halved to preserve the invariant. And, if y is odd, say $2 \times t + 1$, $z + x \times y = z + x \times (2 \times t + 1) = (z + x) + (2 \times x) \times t$. So, z may be replaced by $(z + x)$, x doubled and y halved (using $y \div 2$) to preserve the invariant.

Given the loop invariant, the intermediate assertions can be generated systematically. The annotated program is given in Figure 4.4 (page 163).

Annotated Better multiplication

```

{ $x \times y = prod$ }
{ $0 + x \times y = prod$ }
   $z := 0$ ;
{ $z + x \times y = prod$ }
while  $y \neq 0$  do
  { $z + x \times y = prod \wedge y \neq 0$ }
  if  $odd(y)$  then { $z + x \times y = prod \wedge y \neq 0 \wedge odd(y)$ }
    { $(z + x) + (2 \times x) \times (y \div 2) = prod$ }
     $z := z + x$ 
    { $z + (2 \times x) \times (y \div 2) = prod$ }
  else { $z + x \times y = prod \wedge y \neq 0 \wedge \neg odd(y)$ }
    { $z + (2 \times x) \times (y \div 2) = prod$ }
    skip
    { $z + (2 \times x) \times (y \div 2) = prod$ }
  endif ;
  { $z + (2 \times x) \times (y \div 2) = prod$ }
   $y := y \div 2$ ;
  { $z + (2 \times x) \times y = prod$ }
   $x := 2 \times x$ 
  { $z + x \times y = prod$ }
enddo
{ $z + x \times y = prod \wedge y = 0$ }
{ $z = prod$ }

```

Figure 4.4

Verification condition The only remaining part of the proof is to demonstrate that for every pair of adjacent assertions, p followed by q without any command separating them, $p \Rightarrow q$. The proposition $p \Rightarrow q$ is called a *verification condition*.

A verification condition is a theorem, mostly about the properties of data. Sometimes the proof of a verification condition may appeal to a deep theorem; see, for example, Theorem 5.3 (page 237). Proofs of verification conditions are outside the proper program proof theory. For **Annotated Better multiplication** in Figure 4.4, generate the following verification conditions. Their proofs rely on properties of integer arithmetic.

$$\begin{aligned}x \times y = \text{prod} &\Rightarrow 0 + x \times y = \text{prod} \\z + x \times y = \text{prod} \wedge y \neq 0 \wedge \text{odd}(y) &\Rightarrow (z + x) + (2 \times x) \times (y \div 2) = \text{prod} \\z + x \times y = \text{prod} \wedge y \neq 0 \wedge \neg \text{odd}(y) &\Rightarrow z + (2 \times x) \times (y \div 2) = \text{prod} \\z + x \times y = \text{prod} \wedge y = 0 &\Rightarrow z = \text{prod}\end{aligned}$$

4.6

4.6.1

More on Invariant

Heuristics for Postulating Invariant

This section contains advanced material and may be skipped.

Almost all aspects of program proving are undecidable, that is, there can be no algorithm to prove partial correctness or termination, or even synthesize an appropriate invariant for a loop. There are very few useful heuristics for postulating invariants. The best approach is to understand what is being computed and how it is being computed. With a clear idea about both, a program designer is equipped to construct a formal proof.

Unfortunately, in many cases a program designer may understand the steps of the computation and how the data is being transformed in each step, yet she may find it difficult to translate it into an invariant.⁹

One of the virtues of recursive procedures is that their specification already embodies the invariant. The pre- and postconditions are specified for arbitrary values of the parameters that may arise during a recursive call, not just the values with which the procedure is initially called. In practice, functional programs (which are almost always recursive) are easier to prove. For example, see the proof of *quicksort*, which is mostly recursive, in Section 5.4.

In postulating an invariant, the first question to ask is *not* how the program transforms the data, rather how the program *does not* transform the data. This is a heuristic that we applied for the coffee can problem (Section 4.2.1.1) and the chameleons (Section 4.2.1.2).

9. I had considerable difficulty in postulating a simple invariant for the program in Section 6.5 even though, I thought, I understood it completely.

In this section, I show a result from [Basu and Misra \[1975\]](#) and [Misra \[1977\]](#) about constructing invariants for one class of problems. This result can be directly used in some cases to construct invariants and in other cases to guide the application of heuristics.

Function computed by a loop Consider loop `while  $b(x)$  do  $f$  enddo`, operating on variables x , that is claimed to compute function $F : D \rightarrow D$ over these variables. I show that, under a certain constraint on f , the value of $F(x)$ remains constant, so $F(x) = F(x_0)$ is invariant where x_0 is the initial value of x . Further, this invariant is both necessary and sufficient to prove that the loop computes F .

The specification that the loop computes F is given by:

$$\{x \in D, x = x_0\} \text{ while } b(x) \text{ do } f \text{ enddo } \{x = F(x_0)\}. \quad (\text{A})$$

I place the constraint that $x \in D$ is a loop invariant:

$$\{b(x), x \in D\} f \{x \in D\}. \quad (\text{domain-closure})$$

Theorem 4.1 Assume domain-closure. Then (A) holds iff both of the following conditions hold.

1. $\neg b(x) \wedge x \in D \Rightarrow x = F(x)$,
2. $\{x \in D, b(x), F(x) = c\} f \{F(x) = c\}$, for any c in D .

Proof of the theorem is given in Appendix 4.A.

In many cases, we may only care about the values computed by the loop for some of the variables x' in x . Suppose it is claimed that the final values in x' are $F'(x)$. Given (A) in which F is replaced by F' , the proof of the theorem still applies, so that a necessary invariant is $F'(x) = c$.

As an application of this theorem, consider a loop of the given form that is claimed to compute the greatest common divisor, gcd , of two positive integers m and n in m . The domain D is pairs of positive integers. The specification is, for any positive integer G :

$$\begin{aligned} &\{ (m, n) \in D, m, n = m_0, n_0 \} \\ &\text{while } b(m, n) \text{ do } f \text{ enddo} \\ &\{ m = \text{gcd}(m_0, n_0) \} \end{aligned}$$

Assume domain-closure, so $(m, n) \in D$ is invariant. According to the theorem, we must show that (1) $\neg b(m, n) \Rightarrow m = \text{gcd}(m, n)$, and (2) $\text{gcd}(m, n) = c$ is invariant for any c in D .

Note Function F in Theorem 4.1 has D as both its domain and codomain; so it may seem that f only reads from and writes to variables whose states are given by D . Actually, f can write to variables external to x but not read their values. The values written to external variables is of no concern here, so I have restricted the codomain of F to D .

Note There is a small generalization of Theorem 4.1. Suppose that the precondition of the loop is $x = X$ and the postcondition $x \sim X$, where $\sim$ is some equivalence relation. Then it is necessary and sufficient that $x \sim X$ be an invariant, see Misra [1977]. Theorem 4.1 is a special case where $x \sim y \equiv F(x) = F(y)$.

Limitations of the theorem At first glance, Theorem 4.1 seems to provide a loop invariant magically, all we have to ensure is domain-closure. We need not know anything about how the computation is being performed. In practice, domain-closure is difficult to ensure. To see this, consider the **Better multiplication** program in Figure 4.3 (page 162). The loop is preceded by the assignment $z := 0$. So, D is a set of triples (x, y, z) where the first two components are arbitrary integers and the last component is 0, and $F(x, y, z) = x \times y$. This domain is not closed because any non-zero assignment to z violates domain-closure.

Almost always a loop operates on a domain that is not closed. And just the knowledge of the function on its non-closed domain is insufficient to postulate the invariant.

Another limitation of the theorem is that it is applicable only to computations of functions (or equivalence relations). In many cases, the postcondition of the loop is not functional.

Expanding the domain for closure An invariant is a generalization of the assertion in the precondition of the loop. What is not clear is how to generalize it. Theorem 4.1 suggests how one might approach the generalization problem, which is by expanding the domain so that it is closed and postulating an appropriate generalization of the function on the expanded domain.

As an example, consider the loop that computes $x \times y$ in z in **Better multiplication** program in Figure 4.3 (page 162). The loop body modifies z by adding values to it. Therefore, if the loop indeed computes $x_0 \times y_0$ when started with $x, y, z = x_0, y_0, 0$, it would compute $z_0 + x_0 \times y_0$ if it is started with $x, y, z = x_0, y_0, z_0$. Using the closed domain D of the triples of positive integers, we get the invariant $z + x \times y = c$ for any c in D . This is much simpler than postulating an invariant by pure guesswork.

The last example suggests a general heuristic for expanding the domain. Suppose

$$\{z = 0, X = X_0\} \text{ while } b(x) \text{ do } f \text{ enddo } \{z = F(X_0)\}$$

where X is a set of variables that does not include z . If (1) z never appears in a test nor in any assignment except to itself, (2) the assignments to z are of the form $z := z \oplus g(X)$ for an associative operator $\oplus$, where g is arbitrary, and (3) 0 is the unit of $\oplus$, then $\{z \oplus F(X) = c\} f \{z \oplus F(X) = c\}$, for any constant c in the domain.

Dijkstra often used the heuristic of replacing a constant by a variable in the postcondition to obtain an invariant. That heuristic is a way of creating a closed

domain and the corresponding function, as in the given example by replacing 0 in $0 + x \times y$ by z .

4.6.2 Invariant Strength

A loop has many invariants. Not all invariants are equally useful for its proof. For example, $1 > 0$ is a tautology, therefore an invariant for any loop. But it is hardly the invariant we seek for a specific loop. Less trivially, for the **Better multiplication** program in Figure 4.3 (page 162) we can establish the invariant that $y \leq Y$ given that Y is the initial value of y . However, this invariant provides no information about the relationship between the variables x , y and z ; so, it is inadequate for the proof.

Recall from Section 2.5.3 that predicate p is *stronger* than q if $p \Rightarrow q$. The set of all invariants is partially ordered by the implication operator. The weakest invariant, *true* (or a tautology), is of little use as an invariant. We seek an invariant that is strong enough to allow us to use the loop proof rule. The very strongest predicate, *false*, is an invariant but it holds in no program state; so it would not hold at the start of a loop. Automatic proof systems apply a number of heuristics to deduce an invariant that is strong enough for the given purpose.

4.6.3 Perpetual Truth

A predicate may be true at the start of every loop iteration, but it may not be an invariant. Consider the trivial program in Figure 4.5.

A trivial program

```

x, y := 0, 0;
while x < 10 do
  if y = 0 then x, y := x + 1, 1
  else y := 0
endif
enddo

```

Figure 4.5

The predicate $x \geq y$ holds at the start of every loop iteration. But it is not an invariant because the contract for the “else” clause

$$\{x \geq y \wedge x < 10 \wedge y \neq 0\} \quad y := 0 \quad \{x \geq y\}$$

requires proving the verification condition $(x \geq y \wedge x < 10 \wedge y \neq 0) \Rightarrow (x \geq 0)$. This does not hold because the antecedent does not carry the necessary information about x .

A predicate q is *perpetually true* if $p \Rightarrow q$ for some invariant p . Thus, every invariant is perpetually true, though not conversely. For the example above, show that $x \geq 0 \wedge x \geq y$ is an invariant from which $x \geq y$ follows.¹⁰

Often an invariant is postulated by taking several predicates that are, presumably, perpetually true, combining them and further strengthening them.

4.6.4 Proving Non-termination

Sequential program executions are expected to terminate. So, non-termination is mostly a non-issue for such programs. However, sometimes we like to prove that a certain game will never terminate. For example, in the chameleons problem (see Section 4.2.1.2, page 143) the game does not terminate starting with an arbitrary initial configuration, such as having 1, 2 and 3 chips in the three piles initially. Invariants provide a powerful technique to prove non-termination.

For a program of the form: **initialize; while b do f enddo**, we prove non-termination by showing a loop invariant I such that both I and the loop exit condition $\neg b$ cannot hold simultaneously, that is, $I \wedge \neg b$ is always *false*; equivalently $I \Rightarrow b$. In the case of the chameleons, where p, q and r are the number of chips in the three piles at any point during the game, I proved the invariant:

$$(p - q \equiv^{mod 3} p_0 - q_0) \wedge (q - r \equiv^{mod 3} q_0 - r_0) \wedge (r - p \equiv^{mod 3} r_0 - p_0),$$

where p_0, q_0 and r_0 are the initial number of chips in the corresponding piles. Given the initial condition $p_0, q_0, r_0 = 1, 2, 3$, the invariant I becomes:

$$(p - q \equiv^{mod 3} (-1)) \wedge (q - r \equiv^{mod 3} (-1)) \wedge (r - p \equiv^{mod 3} 2).$$

The game terminates only when any two piles are empty, that is, $\neg b$ is:

$(p - q \equiv^{mod 3} 0) \vee (q - r \equiv^{mod 3} 0) \vee (r - p \equiv^{mod 3} 0)$. It follows that $I \wedge \neg b$ is *false*; so the game does not terminate with these initial values.

An aside: Deadlock in concurrent systems Proving non-termination is vital in concurrent programming. You can appreciate the problem from the following dilemma that faces a beginning car driver. At a 4-way stop if two cars arrive simultaneously on adjacent sides, a driver is required to yield to a car on its right side. What if four cars arrive simultaneously on all four sides? Everyone yields to someone so no one can proceed, leading to what is known as “deadlock”. The scenario sketched here is unlikely, but its theoretical implications should bother some, including computer science students. Beginning drivers soon learn that the deadlock is broken by aggressive driving!

10. Some authors use the term “invariant” for “perpetually true” and “inductive invariant” for what we call “invariant”.

The problem arises far more frequently in computer-based concurrent systems, sometimes with disastrous effect. Consider a situation where multiple users demand certain resources from a resource manager on an ongoing basis to continue with their computations. If the resource manager does not adopt any policy for resource allocation, say allocating available resource to any user who demands it, the system is likely to enter a state of deadlock. Consider a scenario where user X demands resource x , Y demands resource y , X is using y and Y is using x . No user can be granted the resource it wants, so none of them can proceed. Termination, in this case deadlock, is a most undesirable state for a concurrent system.

I show two small examples of non-termination in sequential programs, both in connection with games.

4.6.4.1 Coloring Cells in a Square Grid

Given is a 10×10 grid consisting of 100 cells. Two cells are neighbors if they share a side. Initially 9 cells are arbitrarily chosen and colored, and all other cells are left uncolored. A *move* colors a cell provided it has two colored neighboring cells. Prove or disprove that all cells can be colored. This problem is described in [Dijkstra \[1995\]](#).

Each cell has four sides and each of these sides is *incident* on that cell. Then each side, except those on the boundary, is incident on exactly two cells. Call a side *open* if it is incident on exactly one colored cell. A move “closes” at least two open sides and creates at most two open sides, so the number of open sides does not increase.

Initially, 9 cells are colored, which can create at most 36 open sides. Therefore, the number of open sides remains at most 36. If all cells are colored, the sides on the boundary, all 40 of them, are open. So, such a coloring is not possible. We have used the invariant that the number of open sides is at most 36, and the termination condition that the number of open sides is 40.

4.6.4.2 15-Puzzle

Description This puzzle became famous in mid to late 19th century because Sam Loyd, who falsely claimed to be its inventor, offered a prize of \$1,000 to anyone who could solve it. The puzzle is played on a 4×4 board, each position of which, except one, contains a *tile* numbered 1 through 15 (see Table 4.1, page 170). The position shown as — has no tile, that is, it is *empty*.

A *move* transfers a tile to the empty position from an adjacent position, either horizontally or vertically, thus leaving the tile’s original position empty. A possible sequence of three moves starting from the configuration in Table 4.1(a) is shown in Table 4.2.

Table 4.1 Initial and final configurations in the 15-Puzzle

01	02	03	04	01	02	03	04
05	06	07	08	05	06	07	08
09	10	11	12	09	10	11	12
13	15	14	—	13	14	15	—
(a) Initial				(b) Final			

Table 4.2 Configurations after a few moves

01	02	03	04	01	02	03	04	01	02	03	04
05	06	07	08	05	06	07	08	05	06	07	08
09	10	11	—	09	10	—	11	09	—	10	11
13	15	14	12	13	15	14	12	13	15	14	12
(a) Vertical move				(b) Horizontal move				(c) Horizontal move			

The tiles are initially placed in the configuration shown in Table 4.1(a), where tiles 14 and 15 are out of order in the last row. The goal is to transform the board through a sequence of moves to the configuration shown in Table 4.1(b), where the tiles are in order.

Proof of non-termination There is no solution to this problem. The proof uses an invariant that is not easy to see.

First, convert a configuration to a sequence: start by reading the numbers from left to right along the first row, then right to left in the second row, next left to right in the third row and, finally right to left in the fourth row. The initial configuration in Table 4.1(a) yields: $S_i = \langle 01, 02, 03, 04, 08, 07, 06, 05, 09, 10, 11, 12, \text{—}, 14, 15, 13 \rangle$, and the final configuration in Table 4.1(b) yields $S_f = \langle 01, 02, 03, 04, 08, 07, 06, 05, 09, 10, 11, 12, \text{—}, 15, 14, 13 \rangle$.

A pair of distinct integer values (x, y) in a sequence where neither x nor y is empty is an *inversion* if they are out of order, that is, x occurs before y in the sequence and $x > y$. So, $(08, 07)$, $(08, 06)$, $(15, 13)$ are some of the inversions in S_i . I introduced the notion of inversion in Section 3.4.1 (page 101) in connection with permuting a sequence using transpositions.

Define the *parity* of a sequence to be even or odd depending on the number of inversions in it. That the final configuration cannot be reached from the initial configuration follows from Observation 4.1.

Observation 4.1 (1) The initial and final configurations, S_i and S_f , have different parity.
(2) Each move preserves parity.

Proof.

(1) Transposition of tiles 14 and 15 in S_i yields S_f . The inversions in S_i and S_f are identical except that (15, 14) is an inversion in S_f and not in S_i . Therefore, S_i and S_f have different parities.

(2) A horizontal move transposes — with an adjacent value, so it preserves all inversions, and hence the parity. A vertical move converts sequence $S = A—BvC$ to $S' = AvB—C$, or S' to S , by transposing — and v . Any inversion (x, y) in S where neither of x nor y is v is unaffected by the move. Further, if (x, v) or (v, x) is an inversion in S and x is not in B , then it is unaffected by the move. The only possible changes are that *all* inversions of the form (x, v) in S where $x \in B$ will be removed because x does not precede v in S' , and some inversions of the form (v, x) , where $x \in B$, will be created in S' . I show that an even number of changes are made, thus the parities of S and S' are identical.

Let there be i elements smaller than v and j elements larger than v in B . Prior to the move, every element larger than v contributes to an inversion because v succeeds B ; following the move, all these inversions are removed and j inversions are added corresponding to each element that is smaller than v . So, the number of inversions is increased by $j - i$. Crucially, the length of B , $i + j$, is even, a consequence of the way in which a sequence is created from a configuration. Therefore, $j - i = i + j - 2 \cdot i$ is even. ■

4.7 Formal Treatment of Termination

Partial correctness establishes a conditional property: *if* the program terminates, then it meets its specification. Proof of termination is, therefore, essential. For example, **Better multiplication** program in Figure 4.3 (page 162) does not terminate if y is not a natural number, that is, if it is a negative integer or a real number.

A program may not terminate if evaluation of an expression in it does not terminate, say in executing $x := 3/0$. Henceforth, first guarantee that each expression evaluation terminates by proving that an appropriate precondition holds before its evaluation.

It can be shown that a program terminates if (1) evaluation of every expression in an assignment or test terminates, (2) execution of every **while** loop in it terminates, and (3) call to every procedure terminates. These conditions seem intuitively obvious; however, a formal proof, based on induction on the structure of the program, is needed: (1) a program that is a simple command, *skip* or assignment, terminates given that expression evaluations terminate, (2) a conditional command **if** b **then** f **else** g **endif** terminates because evaluation of b terminates and executions of f and g terminate, inductively, (3) a loop **while** b **do** f **enddo** terminates because

evaluation of b terminates, by assumptions, f terminates inductively and the main loop by assumption, and (4) a procedure call terminates by assumption. Therefore, it is sufficient to develop the proof theory for termination of procedures and loops, assuming that all expression evaluations terminate.

As explained in Section 4.2.2, associate a variant function (i.e., a metric), M , with a loop **while** b **do** f **enddo** so that each iteration of the loop decreases M . Here, M is a function from the program states to a well-founded set with order relation $<$ along which M decreases, and M cannot decrease indefinitely because of well-foundedness. The formal termination condition¹¹ is given below using the loop invariant p :

$$\{p \wedge b \wedge M = m\} f \{ \neg b \vee M < m \}.$$

Often, the metric is a natural number. Then with initial value n of the metric the number of loop iterations is at most n because each iteration decreases the metric by at least one. In many cases, a well-founded lexicographic order (Section 3.5.1.2, page 117) is used as the metric.

For termination of procedure calls, assume that each non-recursive call terminates. For a recursive procedure P , show a well-founded set $(W, \prec)$ so that (1) the values of the parameters and non-local variables accessed by the procedure, x , in every call to P are from W , and (2) for every recursive call from $P(x)$ to $P(x')$, $x' \prec x$. This ensures that the depth of procedure calls is finite.

I show a number of small examples next, not because they are important in practice but because they illustrate the termination problem clearly and require, possibly, unconventional metrics. Several longer examples are treated in Chapter 5.

4.7.1 A Variation of the Coffee Can Problem

Consider the following variation of the coffee can problem (Section 4.2.1.1, page 142). Randomly select a single bean from the can. If it is black, throw it out. If it is white, throw it out and add an arbitrary finite number of black beans (possibly 0) to the can (enough extra black beans are available). The problem is to determine if the game always terminates, that is, if the can eventually becomes empty.

Termination is not obvious because there is no bound on the number of beans in the can, and each step throws out just one bean. Therefore, we cannot put any bound on the number of steps as a specific function of the initial number of beans in the can, a consequence of allowing an arbitrary number of black beans to be

11. This proof rule appears in Owicki and Gries [1976]. They had restricted the well-founded set to natural numbers.

added in some steps.¹² In programming terms, a non-deterministic assignment for the number of black beans has been made from an *infinite* set.

Let w be the number of white beans and b the black beans in the can at any point during the game. A step either decreases b while preserving w , or decreases w and, possibly, modifies b . Thus, the pair (w, b) decreases lexicographically in each step. Since both w and b are non-negative, the lexicographic order is well-founded and the game terminates.

4.7.2 Chameleons Problem Revisited

I showed for the Chameleons game of Section 4.2.1.2 that the game terminates only if some pair of piles are congruent modulo 3 initially. I have not shown that this condition is sufficient, that is, that there is a strategy that leads to termination under this condition. Now, I present a strategy.

As before, let the piles be P, Q and R and the number of chips in them be p, q and r , respectively. Suppose, initially, $p \equiv q \pmod{3}$, and, without loss in generality, assume that $p \leq q$. The strategy is to decrease q in each step while preserving $p \leq q$. Then, eventually, $q = 0$, and at that point, from $p \leq q$, $p = 0$ as well and the game terminates.

Consider a point when at least two of the piles are non-empty. If $p = 0$, then move chips from Q and R to P , otherwise (if $p \neq 0$) move chips from P and Q to R . Thus, each step decreases q . I now show, informally, that $p \leq q$ is preserved.

Initially, $p \equiv q \pmod{3}$, and I have shown in Section 4.2.1.2 (page 143) that $p \equiv q \pmod{3}$ is preserved in each step. Therefore, $p \equiv q \pmod{3}$ is invariant. If $p = 0$, from the assumption that at least two of the piles are non-empty, q is non-zero. Then, from $p \equiv q \pmod{3}$, $q \geq 3$. Moving chips from Q and R to P results in $p = 2$ and $q \geq 2$, thus ensuring $p \leq q$. If $p \neq 0$, moving chips from P and Q to R decreases both p and q by 1 each, thus preserving $p \leq q$. I formalize these arguments in Appendix 4.B (page 201).

4.7.3 Pairing Points with Non-intersecting Lines

We saw this problem earlier in Section 2.8.1.2 (page 60). I proved that given an even number of points in a plane, no three points on a line, it is possible to pair points in such a way that the line segments connecting the paired points are non-intersecting. The proof, by contradiction, showed that the pairing that has the minimum combined length of all the line segments is intersection-free.

12. [Hehner \[2022, Exercise 320, p. 72\]](#), shows how to prove termination using a natural number-based metric. His idea is to postulate the existence of a function on the state of the can and show that the function value decreases, without explicitly showing a specific function.

In this section, I give a constructive procedure to arrive at some pairing that is intersection-free, not necessarily one that minimizes the combined length of all the line segments.

I showed in Section 2.8.1.2 that if any two line segments intersect, then the line segments can be redirected to form a pairing of strictly smaller length. Start with any pairing and apply this redirection step repeatedly until the pairing is intersection-free.

To show that this procedure terminates, we may be tempted to use the length of a pairing as the metric. This is incorrect because the length is a non-negative real number and non-negative real numbers are *not* well-founded under the standard ordering; so, there is a possibility that an infinite sequence of pairings may be constructed, each smaller in length than the previous one. Fortunately, the number of pairings is finite for a finite number of points.

Formally, let c be the combined length of the line segments of the pairing at any step and $np(c)$ be the number of pairings with length less than or equal to c . Then, in Section 2.8.1.2, I have proved the contract:

$\{np(c) = M, \text{ the pairing has a crossing}\} \text{ redirection step } \{np(c) < M\}$.

Metric $np(c)$ is well-founded because the total number of pairings is finite. Therefore, repeated redirection terminates.

4.7.4 A Problem on Matrices

The following problem uses a non-obvious metric for proof of termination. Given is a matrix of integers. Call a row or column a *line* and the sum of its elements the *line sum*. A *toggle* operation on a line replaces each element x of the line by $-x$. Devise a strategy so that every line sum becomes non-negative by suitably applying some number of toggles.¹³

The following strategy is suggested: pick any line whose sum is negative, if there is any, and apply toggle to it. If the program terminates, there is no line with negative sum (otherwise, the program will not terminate). So, it remains to prove termination.

Let s be the sum of all elements of the matrix at any step. A toggle operation replaces a line having a negative sum by one with a positive sum; so, s strictly increases. Now we need to show that s cannot increase without bound. The maximum value of each matrix element is its absolute value; so, the maximum possible value of s is the sum of the absolute values of all elements. Formally, let N be the sum of the absolute values of all matrix elements. We have shown that the metric $N - s$, which is non-negative, decreases in each step, thus proving termination.

13. This example was shown to me by Carroll Morgan.

4.8 Reasoning about Performance of Algorithms

I develop some of the basic ideas of estimating the *cost* of a program, where cost refers to either the running time or space requirements. The issues of external storage, and transfer time between main memory and disk drives, as well as the use of caches are omitted in the analysis.

The termination argument for a program typically employs an integer-valued metric whose value decreases with execution of each command. So, the metric value is a measure of its running time. But the termination metric is a blunt tool; it merely shows a bound on the number of steps of program execution, not a sharp bound. For example, **Better multiplication** program in Figure 4.3 (page 162) can be proved to terminate for all natural numbers y because the metric y decreases in each step. This gives a bound of y on the number of iterations of the loop, whereas it is possible to prove a sharper bound of $\log_2 y$.

How to describe resource usage Rarely does one need the cost of a program for a specific set of inputs on a specific machine that runs on a specific operating system and compiler. The cost measure would not apply for other inputs nor when the platform is upgraded. First, we are interested in program behavior for *all* inputs of a certain size. Different inputs of the same size often incur different costs; so, almost always the “worst-case behavior” of a program for a certain input size is of interest. Consequently, our goal is to find a function over input size that describes the worst-case behavior of a program for all inputs of that size. We expect this function to be monotonic in input size because larger inputs should not incur lower costs.

Next, different machines consume different amounts of resources for the same program and input. A supercomputer may be a million, or billion, times faster than your laptop. Just as we describe high-level programs in a machine-independent fashion, we should also attempt to describe their performance without reference to specific hardware. Fortunately, random access computers differ in their speeds by constant factors, even though the constant factor may be a million or a billion. So, we use a cost function that describes the worst-case behavior up to some constant factor. Space usage is, typically, independent of machines (though a machine with limited memory may have to use external storage devices and thus incur a penalty in running time).

What about the relative costs of operations on the same computer? We know that multiplication consumes more time than addition, and that suggests that we should specify the cost in terms of the speeds of the individual operations. Such analysis would be quite complicated, so, typically, all operations are assumed to take the same amount of time unless there is a need to distinguish them. For example, in computing the cost of long multiplication (see Section 4.10.3.1, page 187),

I take addition and multiplication of *single digits*, rather than arbitrary integers, to take unit time.

Asymptotic growth It is still difficult to describe the cost function of a program for *all* inputs. A program may execute a different algorithm for inputs of different sizes. A sorting program may use an inefficient procedure, like insertion sort that consumes quadratic time, for small input size, and use a more efficient scheme, like merge-sort, for larger inputs. We are typically interested in larger input sizes, so we ignore the behavior for small inputs. The cost function will show the *asymptotic* growth rate for large inputs.

For a cost function like $3 \cdot n^3 - 10^6 \cdot n^2 + n + 10^{30}$, what counts is its highest term n^3 . There is a large fixed cost, 10^{30} , and a large coefficient for the quadratic term. These terms will have noticeable effect for small values of n , but for $n > 10^{10}$ the cubic term will start dominating the result, and shortly afterwards it will dwarf the values of the other terms. One may argue that most of our inputs will be considerably smaller than 10^{10} in size, so the fixed cost will be the dominating term in practice. But, in most cases, we do not see such extreme variation in coefficients, and the highest term starts dominating for relatively moderate input sizes. Yet, there are cases where a theoretically better algorithm is not used in practice because its behavior for practical input sizes is inferior. As an example, Batcher odd-even sort (see Section 4.10.3.2, page 188) is theoretically inferior to a method known as AKS, but Knuth [1998, sec. 5.3.4] concludes that for input size n “Batcher’s method is much better, unless n exceeds the total memory capacity of all computers on earth!”.

Function families The following function families are essential for program analysis: (1) constant, (2) polylog, (3) polynomial and (4) exponential. For input of size n , a constant function, henceforth written as c or d , has a positive real value for all n . A polylog function, written as $(\log_b(n))^c$ or $\log_b^c n$, has base b and exponent c , $b > 1$ and $c > 0$. It turns out that the specific value of b is irrelevant in much analysis as long as it is greater than 1. A polynomial function is of the form n^c , $c > 0$. An exponential function is of the form c^n , $n > 1$. Most cost functions can be expressed as a combination of these functions. In the given sequence of function families, the asymptotic growth rate of any function in a family is higher than that of any in a preceding family, for example, any polynomial function has a lower asymptotic growth rate than any exponential function. Further, within each family, other than constants, a function with higher value of c grows faster.

Four sample functions over argument x are shown in Figure 4.6. The three polynomial functions $1.3x^2 + 3 - \log x$, $0.4x^{2.37} + 0.4x + 50$ and $0.15x^3$ have different growth rates; the ones with higher exponent of x have higher rates of growth, so $0.15x^3$ grows faster than $0.4x^{2.37} + 0.4x + 50$, which, in turn, grows faster than $1.3x^2 + 3 - \log x$. Each of these functions grows slower than $0.02e^x$, an exponential function.

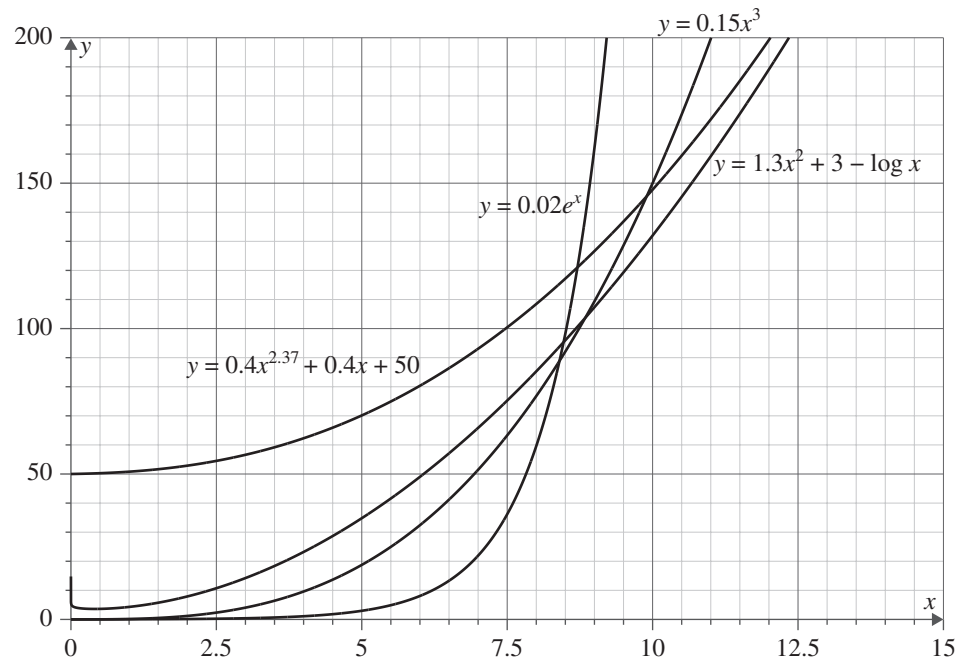


Figure 4.6 Functions with different growth rates.

Amortized cost The worst-case cost of a program for large inputs has proved to be a very useful measure. Sometimes the expected cost can be calculated, and that is also a useful measure. Another measure of cost, called *amortized cost*, is often useful for databases that execute a sequence of procedure (or method) calls. The internal data structure is optimized after each call in anticipation of the next call. We will see an important example of this technique for the Union-Find algorithm in Section 6.7.2 (page 305). Instead of measuring the worst-case cost per call, the total cost over a certain number of calls, the amortized cost, is used in such cases.

Easy versus hard problems A problem is called *easy*, or *tractable*, if it has a polynomial algorithm for its solution, and *hard*, or *intractable*, otherwise. The distinction is obeyed in all models of deterministic computation: an algorithm that runs in polynomial time on one such model runs in polynomial time on all others,¹⁴ even on a Turing machine that does not have random access memory. Therefore, conversely, all exponential algorithms are exponential for all computation models.

There are some problems, usually contrived, for which all algorithms have a very high exponent in their polynomials, so they are not really easy in practice. And there are exponential algorithms that are only moderately expensive for the input sizes of practical interest. But in an overwhelming number of cases, the distinction between easy and hard has proved to be sharp in practice.

A problem of great theoretical and practical importance is SAT-solving, to determine if a given boolean expression is satisfiable (see Section 2.5.4, page 44). SAT-solving belongs to a class of problems known as *NP*-complete, which includes problems of practical importance from a variety of fields. No polynomial algorithm has been found for any *NP*-complete problem, nor has it been proved that such an algorithm does not exist. All known algorithms for these problems have exponential running time. What is known is that all *NP*-complete problems are in the same boat; if there is a polynomial algorithm for one there are for all.

Recent years have seen great advances in SAT-solving in practice. SAT-solvers, like Grasp and zChaff, can solve boolean expressions with millions of variables, unthinkable 20 years ago. SAT-solving is an outstanding example of a problem whose worst-case behavior is very likely exponential, yet in many cases of interest it yields solutions in low polynomial time.

4.9 Order of Functions

I next formalize the notion of asymptotic growth of functions that was described earlier. I consider functions from positive integers to positive integers. The functions of interest are monotone (see Section 2.2.5, page 22) and total over the domain of positive integers. Henceforth, f and g denote such functions.

14. There is one serious proposal, quantum computing, as a model of computation that can solve certain problems in polynomial time that (as far as is known) deterministic digital computers cannot. At any moment, the state of a quantum computer is given by a vector of exponentially many complex numbers. Each step of the computation performs a linear transformation of the vector in unit time. While efforts to build them are underway around the world, truly practical quantum computers do not yet exist.

4.9.1 Function Hierarchy

Function f is *at most* g (g *at least* f), written as $f \sqsubseteq g$, if f 's asymptotic growth rate is no higher than that of g . Formally,

$$f \sqsubseteq g \equiv (\exists n_0, c :: (\forall n : n > n_0 : f(n) \leq c \cdot g(n)))$$

The quantity n_0 excludes a finite number of integers beyond which f grows no faster than g and c is a constant coefficient in comparing their growth rates. Henceforth, I say “predicate $P(n)$ holds for all large n ” to mean there exists an n_0 such that $P(n)$ holds for all $n, n > n_0$. So, $f \sqsubseteq g$ means $f(n) \leq c \cdot g(n)$ for all large n and some $c, c > 0$.

Observe that $\sqsubseteq$ is a reflexive and transitive relation, though it is not a partial order because $f \sqsubseteq g \wedge g \sqsubseteq f$ does not mean that $f = g$. Write $f \doteq g$ for $f \sqsubseteq g \wedge g \sqsubseteq f$.

Terminology I apply arithmetic operators on functions to yield functions with their intended meaning. So, $2 \cdot f$ or $f + g$ denote functions where $(2 \cdot f)(n) = 2 \cdot (f(n))$ and $(f + g)(n) = f(n) + g(n)$.

Basic facts about $\sqsubseteq$ relation In comparing functions using $\sqsubseteq$, we can ignore the constant coefficients of the terms as well as any additive constants. So $c \cdot f + d \doteq f$ for constants c and $d, c > 0$. Consequently, the exact base of logarithms are unimportant because $\log_b n = \frac{\log_a n}{\log_a b}$; thus switching from base a to b changes the value of $\log n$ by a constant multiplicative factor.

We can also ignore all but the “dominant” term, that is, if $f \sqsubseteq g$ then $f + g \doteq g$. This follows directly from the definition: $f \sqsubseteq g$ means that there exists a c such that for all large $n, f(n) \leq c \cdot g(n)$. Then, for all large $n, (f + g)(n) = f(n) + g(n) \leq c \cdot g(n) + g(n) = (1 + c) \cdot g(n)$. And, conversely, $g \sqsubseteq f + g$. For instance, given $f(n) = 3n^2 + 5n + 2 \cdot \log n + 10^6$ and $g(n) = n^2$, we can assert $f \doteq g$.

Another minor observation is that if $f(n) \leq g(n)$ for all large n and σ is a monotonic function, then $\sigma \circ f \sqsubseteq \sigma \circ g$, where $\sigma \circ f$ is the composition of σ with f . Since σ is monotonic, $\sigma(f(n)) \leq \sigma(g(n))$, that is, $(\sigma \circ f)(n) \leq (\sigma \circ g)(n)$, so, $\sigma \circ f \sqsubseteq \sigma \circ g$. As a simple application, I prove that $\log \log \sqsubseteq \log$. Start with $\log n \leq n$ for large n , and then apply the monotonic function $\log$ to both sides.

Lemma 4.1 Let $\star$ be a binary arithmetic operator that is monotonic in both arguments and that *semi-distributes* over multiplication, so that $(d \cdot x) \star (d \cdot y) \leq d \cdot (x \star y)$ for any d, x and y . Then $f \sqsubseteq f'$ and $g \sqsubseteq g'$ implies $f \star g \sqsubseteq f' \star g'$.

Proof. Suppose $f \sqsubseteq f'$. I first show that for any $g, f \star g \sqsubseteq f' \star g$. For any large n :

$$\begin{aligned} & (f \star g)(n) \\ = & \{\text{definition of binary function composition}\} \end{aligned}$$

$$\begin{aligned}
& f(n) \star g(n) \\
\leq & \{f(n) \leq c \cdot f'(n), \text{ for some constant } c, c > 0, \text{ for all large } n\} \\
& c \cdot f'(n) \star g(n) \\
\leq & \{\text{let } d = \max(1, c). \text{ Operator } \star \text{ is monotone in both arguments}\} \\
& d \cdot f'(n) \star d \cdot g(n) \\
\leq & \{\star \text{ semi-distributes over multiplication}\} \\
& d \cdot (f'(n) \star g(n)) \\
= & \{\text{definition of binary function composition}\} \\
& d \cdot ((f' \star g)(n))
\end{aligned}$$

From the definition of $\sqsubseteq$, $f \star g \sqsubseteq f' \star g$. Similarly, if $g \sqsubseteq g'$, then $f' \star g \sqsubseteq f' \star g'$. So, given $f \sqsubseteq f'$ and $g \sqsubseteq g'$, $f \star g \sqsubseteq f' \star g \sqsubseteq f' \star g'$. ■

Examples of binary $\star$, above, are: $+$, $\min$ and $\max$. Multiplication, $\times$, does not always satisfy $(d \cdot x) \times (d \cdot y) \leq d \cdot (x \times y)$, but the proof steps in Lemma 4.1 can be modified slightly, skipping the introduction of d , to arrive at the same conclusion: $c \cdot f'(n) \times g(n) = c \cdot (f'(n) \times g(n))$. Note that exponentiation does not obey semi-distributivity: $(d \cdot x)^{d \cdot y} \leq d \cdot x^y$ does not hold in general.

Corollary 4.1 Let σ be an m -ary arithmetic operator that is monotonic in each argument and that semi-distributes over multiplication for each argument. Then $f_i \sqsubseteq f'_i$, for $1 \leq i \leq m$, implies $\sigma \circ (f_1, f_2, \dots, f_m) \sqsubseteq \sigma \circ (f'_1, f'_2, \dots, f'_m)$.

Proof. Similar to that of the Lemma 4.1. ■

We can now establish a hierarchy among functions starting with a few basic inequalities. For constants b and c , where $b > 1$ and $c > 0$, and large n , observe that $c \leq \log n \leq n \leq b^n$.

Theorem 4.2 Let b, c and d be constants where $b > 1$, $c > 0$ and $d > 0$. Below, i is any positive integer, $\text{poly}_c(n) = n^c$ and $\text{exp}_b(n) = b^n$.

1. $c \doteq d$.
2. $c \sqsubseteq \overbrace{\log \log \dots \log}^{i+1} \sqsubseteq \overbrace{\log \log \dots \log}^i$.
3. $\log^i \sqsubseteq \log^{i+1} \sqsubseteq \text{poly}_c$.
4. $\text{poly}_c \sqsubseteq \text{poly}_d$, where $c \leq d$.
5. $\text{poly}_c \sqsubseteq \text{exp}_b \sqsubseteq \text{exp}_d$, for all c and $b < d$.

In summary: constant $\sqsubseteq$ polylog $\sqsubseteq$ polynomial $\sqsubseteq$ exponential. ■

4.9.2 Function Order

For a function g , introduce three sets:

$\mathcal{O}(g) = \{f \mid f \sqsubseteq g\}$, functions that are at most g .

$\Omega(g) = \{f \mid g \sqsubseteq f\}$, functions that are at least g .

$\Theta(g) = \{f \mid f \dot{=} g\}$, that is, $\mathcal{O}(g) \cap \Omega(g)$, functions with same growth rate as g .

A pictorial depiction of these sets is given in Figure 4.7. Here $\mathcal{O}(g)$ and $\Omega(g)$ are the sets of functions below and above the curves, respectively, next to these symbols, and $\Theta(g)$ is the set of functions that is above $\Omega(g)$ and below $\mathcal{O}(g)$.

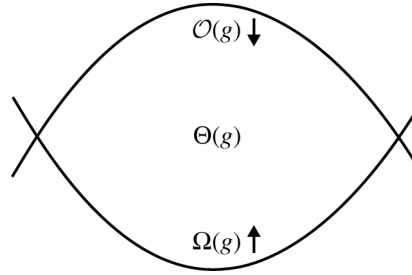


Figure 4.7 Function classification with respect to the growth of g .

As examples, let $f(n) = c \cdot n^2 + c'$ and $g(n) = c \cdot n \log n + c'$, for some constants c and c' and large n . Among sorting algorithms, the worst-case running times of insertion sort, quicksort and merge-sort belong to $\Theta(f)$, $\Theta(f)$ and $\Theta(g)$, respectively. The expected running time for quicksort is $\Theta(g)$.

Example 4.10

1. $\mathcal{O}(1)$ is the set of functions whose values are constant for all large argument values.
2. $\mathcal{O}(n^{2.81}) \subseteq \mathcal{O}(n^3)$. And $\Omega(n^3) \subseteq \Omega(n^{2.81})$.
3. $\Theta(3 \cdot x^2 + 2 \cdot \log x + 5) = \Theta(x^2)$
4. $n^e \in \mathcal{O}(e^n)$, where e is the base of natural logarithm.
5. $n^{10^{10}} \in \mathcal{O}(2^n)$.

Properties of order Several properties of $\mathcal{O}$ follow from the properties of $\sqsubseteq$:

1. $f \in \mathcal{O}(f)$
2. $\mathcal{O}(\mathcal{O}(f)) = \mathcal{O}(f)$
3. $f \sqsubseteq g \Rightarrow \mathcal{O}(f) \subseteq \mathcal{O}(g)$
4. $f \in \mathcal{O}(f'), g \in \mathcal{O}(g') \Rightarrow (f \star g) \in \mathcal{O}(f' \star g')$,
where $\star$ is any binary monotonic operator over integers that semi-distributes over multiplication (see Lemma 4.1, page 179).

Properties (1) to (3) define a *closure* operator.

Loose, conventional terminology A conventional way of describing the running time of an algorithm is that “it runs in $\mathcal{O}(n^2)$ time”. Several things are wrong with this statement; let us enumerate them. First, n^2 is not a function. What is meant, presumably, is that the running time belongs to $\mathcal{O}(T)$ where T is the function defined by $T(n) = n^2$. Second, it is customary to write the cost function *is* or *equals* $\mathcal{O}(n^2)$ rather than that the cost function *belongs to* $\mathcal{O}(n^2)$; it is a type violation to say that $n^{1.5} = \mathcal{O}(n^2)$, equating an element with a set. Each of the function orders, $\mathcal{O}$, Ω and Θ , is a *set* of functions. Third, it is common to see an equation of the form $T(n) = T(n/2) + \Theta(n^k)$ where the desired meaning is $T(n) = T(n/2) + f(n)$ where $f(n) \in \Theta(n^k)$. Further, often $T(n) \leq T(n/2) + f(n)$ holds rather than $T(n) = T(n/2) + f(n)$. This is because such recurrences arise in algorithms in which a problem of size n is decomposed into two subproblems of size $n/2$ that are solved recursively, and where the cost of decomposition is *at most* $f(n)$. Finally (yes, the confession is at its end), it is customary to use $\mathcal{O}$ even when a sharper bound with Θ could have been used.

This terminology has pervaded computer science. In its defense, it is a compact way of describing the cost, avoiding introduction of a function symbol, when it is understood that the input size is n , or some other designated variable, and $f(n) = \mathcal{O}(n^2)$ really means $f(n) \in \mathcal{O}(n^2)$. I bow to this convention and adopt this terminology: we say that the cost $T(n)$ of an algorithm for input size n is $\mathcal{O}(n^2)$, or simply, the cost of the algorithm is $\mathcal{O}(n^2)$, and write $T(n) = \mathcal{O}(n^2)$ with the desired meaning. Never write $\mathcal{O}(n^2) = T(n)$; the order of conventional writing treats $=$ as $\in$. And, I will be guilty of writing $T(n) = T(n/2) + f(n)$ when the exact statement should be $T(n) \leq T(n/2) + f(n)$, because the goal is merely to establish a loose $\mathcal{O}$ bound.

4.10 Recurrence Relations

A *recurrence relation*, or recurrence equation, is an arithmetic equation of the form $f(n) = g(n)$, where f is an unknown function and g an expression that includes known and unknown function symbols including f , constants and arithmetic operators. An example is $T(n) = T(n-1) + n$, where T is the unknown function over n . The goal is to determine T so that the equation is satisfied for all large n . Typically, the exact values of T for small constants are given, as in $T(1) = 1$. In this case the exact solution for $T(n)$ is $\frac{n \times (n+1)}{2}$, which can be verified directly for $T(1)$ and then by substituting the expression in both sides of the recurrence relation.

Recurrence relations are used to estimate the costs of algorithms, particularly when an algorithm is expressed recursively. Recursive programming is the subject of Chapter 7. A typical recursive program divides the given problem into subproblems, solves the subproblems recursively and combines their solutions to arrive at

the solution of the original problem. This approach is also known as “Divide and Conquer” in the literature on algorithms. It would be more appropriate to call the approach “Divide, Conquer and Combine”, since most problems require first a division of a problem into subproblems, next recursive solutions of the subproblems, and finally combinations of the solutions of the subproblems for the solution of the original problem.¹⁵

The cost of an algorithm for a problem of size n , postulated to be $T(n)$, can be expressed as a function of the costs of the subproblems, of the form $T(m)$ where $m < n$, and the cost of dividing into subproblems and combining their solutions. Given the recurrence relation $T(n) = 2T(n-1) + cn$ for some constant c , we may surmise that the original problem of size n is divided into two problems each of size $n-1$, solution of each subproblem costs $T(n-1)$, and the division and combination of the problems has a cost proportional to n .

Typical recurrence relations do not include exact costs for the division and combination steps. Instead, the order of the cost, such as $\mathcal{O}(1)$ or $\mathcal{O}(n)$, are used. The expected solution is also given using the order notation. Sometimes, the recurrence relation may be an inequality rather than equation, and it is expected that only an upper bound on the cost is to be estimated. It is customary to omit the exact cost for small values of the input size since it would be some constant.

This section includes a number of examples of recurrence relations and methods for their solutions. A few longer, more involved examples are shown in Section 4.10.3 (page 187).

4.10.1 Smaller Examples

4.10.1.1 Searching Sorted and Unsorted Lists

Consider searching a sorted list of n items for a particular value *key*. The well-known binary search (Section 5.1) solves the problem by comparing *key* with the element e that is around the middle of the list. If $key = e$, we are done. If $key < e$, then *key* can only be among the elements smaller than e , and, similarly, if $key > e$, it can only be among the elements larger than e . In each case, the original list is reduced to about half of its original size. As we are looking for the worst-case behavior of the algorithm, we only consider the cost when *key* is not in the given list.

We can write the recurrence relation, using $T(n)$ for the search time for a list of length n , as $T(n) \leq T(n/2) + c$, for some constant c . Do not worry about n not being divisible by 2; we can ignore such questions because we are looking for the order

15. “Divide and Conquer” usually refers to algorithms in which each recursion step considers two or more subproblems. When there is just one subproblem, recursion is still useful, though it can be trivially eliminated in the case of tail recursion.

of the solution, not its exact value. The base case is given by $T(1) = 1$. Verify that $T(n) = \mathcal{O}(\log n)$.

Now consider searching an unsorted list by the same technique. Suppose the first comparison is made with the element at position k , $k \geq 0$. We can no longer eliminate half of the list after the first comparison, so we have to search both halves. The resulting recurrence relation is $T(n) = T(k) + T(n - k - 1) + c$. Verify that $T(n) = \mathcal{O}(n)$.

4.10.1.2 Merge-Sort

Merge-sort operates on a list as follows: (1) divide the list into two halves of roughly equal lengths and sort each half recursively, then (2) merge the two sorted halves to obtain the desired sorted list. I consider a variation of this classic scheme: divide the original list into k segments, $k \geq 2$, of roughly equal lengths, sort each segment recursively and then merge the sorted segments. Short lists, say of length less than 5, are sorted using some simpler technique, like insertion sort. Assume that merging k sorted lists of combined length n takes $\Theta(n \log k)$ time. This claim can be justified by using a heap data structure of k items (see Section 5.5.2, page 216).

Let $T(n)$ be the cost of merge-sort. We have:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{if } n < 5 \\ k \times T(n/k) + \Theta(n \log k) & \text{otherwise} \end{cases}$$

Solving with $k = 2$, the classical merge-sort, we get $T(n) = \mathcal{O}(n \log n)$. Solving with $k = \sqrt{n}$, again $T(n) = \mathcal{O}(n \log n)$. I show that for $k = \log n$, the equation $T(n) = (\log n) \times T(n/\log n) + n \times \log(\log n)$ has the same solution, $\mathcal{O}(n \log n)$.

$$\begin{aligned} & (\log n) \times T(n/\log n) + n \times \log(\log n) \\ = & \{\text{substitute } n \log n \text{ for } T(n)\} \\ & (\log n) \times (n/\log n) \log(n/\log n) + n \times \log \log n \\ = & \{\text{arithmetic}\} \\ & n \times (\log n - \log \log n) + n \log \log n \\ = & \{\text{arithmetic}\} \\ & n \cdot \log n \end{aligned}$$

The reader can verify that if k is taken to be $n/2$ so that the original list is divided into segments of length 2, then sorted and merged, the cost would still be $\mathcal{O}(n \log n)$. Even though the costs are asymptotically equal in all cases, they differ in their constant coefficients, so, in practice division into two segments is always preferable.¹⁶

16. In external sorting, this claim is not true because the cost of picking the least element among k sources is small and the cost of fetching an element from secondary memory is large.

4.10.2 Solving Recurrence Relations

We have seen a few recurrence relations and their solutions in the last section. The solutions were postulated and then verified. This is a useful method, particularly after you have seen a variety of patterns and observed the similarities in their solutions. I propose a few general techniques for solving recurrence relations.

4.10.2.1 Interpolation

Given a recurrence relation of the form $T(n) = f(n)$, interpolation is applied in the following sequence: (1) rewrite the equation by removing all order operators, $\mathcal{O}$, Θ and Ω , instead introducing unknown constants in their place as given by the definitions of the order operators, (2) postulate a function g for T and replace $T(m)$ by $g(m)$ in both sides of the equation, and (3) determine if the unknown constants can be assigned values so that the equation is satisfiable for all large n . If the last step succeeds, we have guessed correctly. If it is impossible to satisfy the equation for large n , say the left side of the equation exceeds the right side, our guess was too high, so choose a smaller function (one that has asymptotically smaller values); if the left side is smaller choose a larger function. In most cases, the postulated function is a combination of polynomial, polylog and exponential functions.

As an example, consider the recurrence relation $T(n) = 2 \cdot T(n/3) + c \cdot n$ for some unknown constant c . Guess a solution, say $d \cdot \log n$; it is clear that this guess is too small because the right side has a $\mathcal{O}(n)$ term. This will become apparent once we start solving the relation. Substituting $d \cdot \log n$ for $T(n)$ in both sides we get, $d \cdot \log n = 2 \cdot d \cdot \log(n/3) + c \cdot n$. The left side is too low because of the $c \cdot n$ term in the right side. Next try $d \cdot n$ for $T(n)$. The resulting equation is $d \cdot n = (2/3 \cdot d + c) \cdot n$, that is, $d = 3 \cdot c$, so the equation is satisfiable. Show that $T(n) = 2 \cdot T(n/2) + c \cdot n$ is not solvable with $d \cdot n$ for $T(n)$.

Interpolation is a method for guessing an inductive hypothesis, in this case the form of a function. Interpolation is applicable here because we can compare two functions for “high” or “low”, not just inequality. Interpolation is not generally applicable in inductive proofs because predicates cannot be compared for “high” or “low”; see, however, [McMillan \[2005\]](#) for interpolation techniques applied in automated program verification.

4.10.2.2 Expansion

A useful strategy for solving the recurrence relation $T(n) = f(n)$ is to replace all occurrences of $T(m)$ in the right side by $f(m)$. Continue this for a few steps to observe the emerging pattern that often reveals the desired function. I illustrate with a small example. Consider the recurrence relation $T(n) = 2 \cdot T(n/2) + c \cdot n \log n$.

I show its expansion below. Observe that, replacing n by $n/2^i$, $0 \leq i \leq \lfloor \log n \rfloor$, in the given recurrence we get,

$$\begin{aligned} T(n/2^i) &= 2 \cdot T(n/2^{i+1}) + c \cdot n/2^i \log(n/2^i). \text{ That is,} \\ T(n/2^i) &= 2 \cdot T(n/2^{i+1}) + c \cdot n/2^i \cdot (\log n - i). \end{aligned} \quad (1)$$

$$\begin{aligned} &T(n) \\ &= \{\text{given recurrence relation}\} \\ &\quad 2 \cdot T(n/2) + c \cdot n \log n \\ &= \{\text{with } i = 1 \text{ in (1), } T(n/2) = 2 \cdot T(n/4) + c \cdot n/2 \cdot (\log n - 1)\} \\ &\quad 2 \cdot (2 \cdot T(n/4) + c \cdot n/2 \cdot (\log n - 1)) + c \cdot n \log n \\ &= \{\text{simplifying}\} \\ &\quad 4 \cdot T(n/4) + 2 \cdot c \cdot n \log n - c \cdot n \\ &= \{\text{with } i = 2 \text{ in (1), } T(n/4) = 2 \cdot T(n/8) + c \cdot n/4 \cdot (\log n - 2)\} \\ &\quad 4 \cdot (2 \cdot T(n/8) + c \cdot n/4 \cdot (\log n - 2)) + 2c \cdot n \log n - c \cdot n \\ &= \{\text{simplifying}\} \\ &\quad 8 \cdot T(n/8) + 3 \cdot c \cdot n \log n - 3 \cdot c \cdot n \\ &= \{\text{observe the pattern and conclude}\} \\ &\quad 2^k \cdot T(n/2^k) + k \cdot c \cdot n \log n - k \cdot c \cdot n, \text{ for all } k, 0 \leq k \leq \lfloor \log n \rfloor \end{aligned}$$

Set $k = \lfloor \log n \rfloor$ so that $n/2^k$ reduces to a constant close to 1 so that the term $2^k \cdot T(n/2^k)$ becomes $\mathcal{O}(n)$. The resulting expression becomes $\mathcal{O}(n) + \mathcal{O}(n \log^2 n) = \mathcal{O}(n \log^2 n)$.

In many cases the right side becomes an arithmetic, geometric or harmonic series. The following identities are then useful in reducing those expressions.

$$\text{Arithmetic series: } (+i : 0 \leq i \leq n : i) = \frac{n(n+1)}{2}$$

$$\text{Geometric series: } (+i : 0 \leq i \leq n : x^i) = \frac{x^{n+1} - 1}{x - 1}$$

$$\text{Infinite geometric series: } (+i : 0 \leq i : x^i) = \frac{1}{1-x}, \text{ where } -1 < x < 1$$

$$\text{Harmonic series: } (+i : 0 \leq i \leq n : 1/i) = \ln n + \mathcal{O}(1), \text{ where } \ln \text{ is}$$

“natural” logarithm.

4.10.2.3 Master Theorem

Master Theorem is an important tool for solving recurrence relations. I give a simplified version of the theorem below. A divide and conquer solution to a problem divides a problem of size n into q pieces, each of size n/p . The parameters p and q may not be the same because the division may not partition the subproblems into disjoint pieces. Suppose that the division and later recombination of the solutions of the subproblems incurs a polynomial cost, of the order of $\Theta(n^k)$ for some

non-negative constant k . The cost for smaller values of n are not specified; they are, typically, small constants.

Theorem 4.3 For large n suppose $T(n) = q \cdot T(n/p) + \Theta(n^k)$, where p, q and k are constants with $p > 1, q \geq 1$ and $k \geq 0$. Let $c = \log_p q$. Then,

$$T(n) = \begin{cases} \Theta(n^c) & \text{if } k < c \\ \Theta(n^c \cdot \log n) & \text{if } k = c \\ \Theta(n^k) & \text{if } k > c \end{cases}$$

The theorem yields solutions for $T(n)$ that include only polynomial and polylog but not exponential functions.

Example 4.11 Table 4.3 shows a number of recurrence relations and their solutions using the master theorem.

Table 4.3 Sample cost functions and their solutions

Equation	$c = \log_p q$	k	$T(n) =$
$T(n) = 16 \times T(n/2) + 6 \times n^2$	4	2	$\Theta(n^4)$
$T(n) = 8 \times T(n/2) + 2 \times n^3$	3	3	$\Theta(n^3 \log n)$
$T(n) = 2 \times T(n/2) + 3 \times n^2$	1	2	$\Theta(n^2)$

4.10.3 Divide and Conquer

4.10.3.1 Long Multiplication

How fast can we multiply two n digit numbers? The grade school algorithm multiplies each digit of one with the other and adds up the appropriate results; so it incurs a cost of $\Theta(n^2)$.

A divide and conquer approach may take the left and right halves of each number, multiply them recursively and add them. Specifically, suppose the numbers to be multiplied are given in base b and $n = 2m$. Let the left half of one number be x and the right half x' , so the number is $b^m \cdot x + x'$. Let the other number, similarly, be $b^m \cdot y + y'$. Then the product $(b^m \cdot x + x') \times (b^m \cdot y + y') = b^{2m} \cdot z + b^m \cdot z' + z''$, where $z = x \cdot y$, $z' = x \cdot y' + x' \cdot y$, and $z'' = x' \cdot y'$. Computations of z, z' and z'' involve 4 multiplications of m digit numbers and 3 additions. For numbers given in binary, that is, $b = 2$, multiplication with b^m amounts to left-shifting a number by m positions, which incurs $\Theta(m)$ cost. Using the recurrence $T(n) = 4T(n/2) + \Theta(n)$ for the cost, we get $T(n) = \Theta(n^2)$, no improvement over the grade school method.

Karatsuba's algorithm I present a slightly simplified version of an algorithm due to Karatsuba for computing z, z' and z'' as defined above; it uses only 3 multiplications and a few more additions. Specifically, $z = x \cdot y$ and $z'' = x' \cdot y'$ as before.

But $z' = (x + x')(y + y') - z - z''$. Therefore, the cost is given by the recurrence $T(n) = 3T(n/2) + \Theta(n)$. Using the Master theorem, $T(n) = \Theta(n^{\log_2 3})$. Since $\log_2 3$ is approximately 1.585, this is a significant improvement over the grade school method.

For even better asymptotic algorithms, see [Schönhage and Strassen \[1971\]](#) and [Fürer \[2007\]](#).

4.10.3.2 Batcher Odd–Even Sort

There is an ingenious sorting method due to [Batcher \[1968\]](#) that is of interest in parallel sorting, that is, using several computers that proceed in lock-step fashion. The algorithm is described in Section 8.5.4 (page 479). Here we are only interested in developing and solving recurrence relations for its cost, so I don't prove its correctness. No familiarity with parallel computing is needed for the material explained next.

The list to be sorted is L . Write L_{even} and L_{odd} for its sublists consisting of the elements with even and odd indices in the original list. So, for $L = \langle 7\ 3\ 0\ 9 \rangle$, $L_{\text{even}} = \langle 7\ 0 \rangle$ and $L_{\text{odd}} = \langle 3\ 9 \rangle$. Assume that the length of L is a power of 2, so that its sublists again can be halved if they have more than one element. Such a list is called a *powerlist* in Chapter 8.

Batcher sort involves sorting L_{even} and L_{odd} independently and then merging the sorted lists. Write $B\text{sort}$ and $B\text{merge}$ for these functions below. Function $B\text{merge}'$ is also a merge step, but because of certain special properties of its argument lists it can be executed much faster than a general merge.

For a list of length 1, there is no work to do in $B\text{sort}$. Similarly, $B\text{merge}$ of lists of length 1 involves a single comparison and swap, if needed; so its cost is $\mathcal{O}(1)$. I show the steps of the algorithm for the general case.

```
(1) Bsort(L)::
    (1) sort two halves,  $L_{\text{odd}}$  and  $L_{\text{even}}$ :
        (1.1)  $X := B\text{sort}(L_{\text{odd}})$ ;
        (1.2)  $Y := B\text{sort}(L_{\text{even}})$ ;
    (2)  $B\text{merge}(X, Y)$ 

(2) Bmerge( $X, Y$ )::
    merge halves of  $X$  and  $Y$ :
        (2.1)  $Z = B\text{merge}(X_{\text{odd}}, Y_{\text{even}})$ ;
        (2.2)  $Z' = B\text{merge}(X_{\text{even}}, Y_{\text{odd}})$ ;

(3)  $B\text{merge}'(Z, Z')$ 
```

First, let us compute the number of steps when each of the given steps is executed in order, that is, the algorithm is run sequentially. Execution of $Bmerge'(Z, Z')$ takes $\mathcal{O}(n)$ steps. Writing $S(n)$ for the cost of $Bsort$ and $M(n)$ for $Bmerge$ on lists of length n , we have:

$$\begin{aligned} S(n) &= 2S(n/2) + M(n) \\ M(n) &= 2M(n/2) + \mathcal{O}(n) \end{aligned}$$

From the Master theorem, solution for $M(n)$ is $\mathcal{O}(n \log n)$. For $S(n)$, the recurrence relation is solved as an example under Expansion (Section 4.10.2.2, page 185); $S(n)$ is $\mathcal{O}(n \log^2 n)$.

Now, estimate the running time for parallel computation. Steps (1.1) and (1.2) can be executed in parallel. Also, (2.1) and (2.2) can be executed in parallel. Batcher showed that $Bmerge'(Z, Z')$ can be computed in $\mathcal{O}(1)$ parallel steps. So, we get:

$$\begin{aligned} S(n) &= S(n/2) + M(n) \\ M(n) &= M(n/2) + \mathcal{O}(1) \end{aligned}$$

From the Master theorem, or simply by inspection, $M(n) = \mathcal{O}(\log n)$. Verify that $S(n)$ is $\mathcal{O}(\log^2 n)$.

4.10.3.3 Median-Finding

I describe a beautiful recursive algorithm from Blum et al. [1973] that finds the median of a set of numbers in linear time (median of a set of odd size is an element that has equal number of elements smaller and larger than it). A generalized version finds the k^{th} smallest number, or the element of rank k , in a set of n numbers, $1 \leq k \leq n$, again in linear time. For the rank k element, there are exactly $n - k$ larger numbers and $k - 1$ smaller numbers. In the special cases of $k = 1$ and $k = n$, the rank k element is the minimum and the maximum of the set, respectively; there are simple $\mathcal{O}(n)$ algorithms to solve these problems. For arbitrary k , the simple strategy is to sort the set, at a cost of $\mathcal{O}(n \log n)$, and then pick the k^{th} number of the sorted list. But it is possible to do far better. Function $select(S, k)$ in Figure 4.8 (page 190) returns the element of rank k from set S of at least k elements in $\mathcal{O}(n)$ time. I use $|T|$ for the size of set T in this algorithm.

Estimating the cost I ignore small additive constants in this calculation. Let $T(n)$ be the cost for a set of size n . Step 1 involves taking the median of $n/5$ subsets, each of size 5 or less. Since $T(5)$ is a constant, the cost is proportional to n , that is, $c \cdot n$ for some c . Step 2 incurs a cost of $T(n/5)$ since the size of M is around $n/5$. For step 3, we need to estimate the size of lo and hi . The size of M is around $n/5$ and m is larger than about half of the medians, that is, $n/10$ elements of M . Each median is greater than or equal to 3 items out of 5. So, m is larger than $3 \cdot n/10$ elements. Therefore, the size of hi is at most $7 \cdot n/10$ because it excludes all elements smaller than m .

select (*S*, *k*)

1. If *S* is a small set, say of size 10 or less, find the desired element by sorting the set. Otherwise, use the following steps.
 2. Divide *S* into subsets of 5 elements each and compute the median of each subset. If $|S|$ is not exactly divisible by 5, there is a left-over subset of size less than 5; take its median. Let the set of medians be *M*.
 3. Compute the median of *M*:
 $m := \text{select}(M, \lceil |M|/2 \rceil).$
 4. Partition *S* based on *m*:
 $lo := \{x \mid x \in S, x \leq m\}; \quad hi := \{x \mid x \in S, x > m\}.$
 5. Recursively find the desired element in *lo* or *hi*:
if $k \leq |lo|$ **then** { the rank *k* element of *S* is the rank *k* element of *lo* }
 $\text{select}(lo, k)$
else { the rank *k* element of *S* is the rank $(k - |lo|)$ element of *hi* }
 $\text{select}(hi, k - |lo|)$
endif
-

Figure 4.8

Symmetrically, the size of *lo* is at most $7 \cdot n/10$. So, we get the relation:

$$T(n) \leq c \cdot n + T(n/5) + T(7 \cdot n/10).$$

The Master theorem is not directly applicable here. Verify that $T(n) \leq d \cdot n$ for some constant *d*. You can also arrive at this result using the method of expansion (Section 4.10.2.2, page 185).

4.11 Proving Programs in Practice

This chapter promised to describe a proof theory for (a small fragment of) an imperative programming language. It did not promise that the theory would be easy to use in practice. So, the reader may feel that the theory can be ignored because it's too hard to use today. That is not the message I would like to convey. Regardless of whether one applies the methodology in detail, knowledge of it improves one's approach to programming. A program built carefully with a clear idea of its major invariants is likely to work sooner and be more understandable, maintainable and reliable than a program naively put together and then debugged into some kind of working order.¹⁷

17. A slogan due to Harlan Mills, an IBM fellow now deceased, is "if you play long enough you will put the golf ball in all 18 holes". He was a mentor and the thesis advisor of the author.

When not to verify I am writing this manuscript using a text processor to input the raw text and a document processor to display it in a form in which the figures, formulae and text are shown appropriately. I am not particularly concerned that none of these products have been verified and probably not even tested to the level I would like. I see my input keystrokes being displayed immediately on the screen, and that is about all I expect from a text processor (though I use some of its advanced macro functions to reduce my amount of typing). And the document processor, in spite of sometimes claiming that my correct input is actually incorrect, works. The error reports from many thousands of users usually lead the developers to engage in a virtuous cycle of removing and improving the product. The level of guarantee that I would demand in avionics when I am flying in a plane does not have to be remotely met by these products.

Meeting the ideal goal of completely verifying every piece of software is still well in the future. What we expect from current software is a sufficient level of certainty, not any absolute guarantee. A piece of code that is obviously correct does not need to be verified. The degree of analysis that I apply to the algorithms in this book make me confident of their correctness.

Cost of software bugs Verifying that a program meets its specification is laborious and impossible to complete for many lengthy and complex programs. That is why software bugs are still a problem and an astoundingly expensive problem. A study commissioned by the US Commerce Department in 2003 found that software bugs cost the US economy around \$60 billion annually, probably much more now. Theory of programming has made it possible to reduce errors and also made it possible to design far larger programs.

Bulk versus intricacy It helps to distinguish between the *bulk* and *intricacy* of a program, its size and its complexity.¹⁸ Often the size of a program is taken to be a measure of its intricacy. Actually, a large well-structured program may be easier to understand than an intricate poorly documented program. A key to managing proofs of large programs is to structure a program to consist of a number of small modules (procedures or data abstractions), each of which has a well-defined interface. As I have shown with procedures, the caller of a procedure need not be aware of its implementation details. Intricate programs do require more formal treatment. I use the ideas of program proving for many problems in this book where such a treatment is merited.

18. I eschew the term “complexity” for “intricacy” because complexity already has a technical meaning in the theory of algorithm analysis.

Most software evolves with time. It is extremely laborious and painful to debug software after patches have been applied to it because in many cases the original product has not been sufficiently documented. The documentation of software, by writing the specifications of the major modules, their interactions with the other modules and the key invariants are probably the best gift you can bestow on someone who will eventually maintain it.

Tools for verification The amount of work involved in proving correctness grows much faster than the size of the program. So, computers are often used in this effort. Computers have been used as proof assistants to help human provers from the early days. The science and engineering of program verification has evolved over the years by incorporating better principles for program design and using more capable software tools to assist in verification. Combined with the increasing processor speeds and storage capabilities, significantly larger programs are being proved today.

Automated software tools are used heavily for testing and verification. One particularly important method, called Model Checking, see [Clarke and Emerson \[1982\]](#) and [Queille and Sifakis \[1982\]](#), is especially useful for verifying finite state systems. It is invariably used in hardware verification and for many critical software modules such as for cache coherence protocols. Critical software systems of medium to large size typically undergo some automated verification. However, entirely automatic verifications of very large systems are still out of reach. Their proofs tend to be long, often astronomically long, so that they cannot be completed within any reasonable time even by a supercomputer.

Along with improvements in hardware and automated tools, advances in programming languages and methodologies have played a significant role in making verification more tractable. Many aspects of program optimization, for example, are delegated to compilers, so there is often no need for programmers to indulge in intricate coding to manage heaps and garbage collection, say. Advances in type theory have made it far simpler for programmers to avoid “silly” mistakes. The original theory of program verification has been considerably enhanced, particularly with the introduction of programming logics, for example, temporal and separation logics.

A coordinated international effort for automated verification, Verified Software Initiative (VSI), was started around 2000 and was spearheaded by Tony Hoare, nearly three decades after publication of his seminal paper “An Axiomatic Basis of Computer Programming”. It was then felt that the available theories and capabilities of the machines were sufficient to embark on ambitious, industrial-level automated verification. There have been substantial achievements since then, both

in the variety and capabilities of the tools and their applications on software of considerable size and complexity.

The next phase of VSI starts with considerably more theory and more sophisticated verification systems. The truly important factor, however, is the number and quality of researchers who are ready to address the next set of problems, and the awareness among software manufactures that this is a problem of fundamental importance for the growth of the industry.

We expect the current trends in better automated tools and programming languages to continue so that we may one day use program provers as routinely as programmers use debugging tools today; see [Hoare and Misra \[2009\]](#) and the associated special issue of the *ACM Computing Surveys* on automated verification for discussions of some of the technical issues.

4.12 Exercises

For each problem that asks for a proof, develop both partial correctness and total correctness proofs.

1. What is p in: $\{p\} s := \text{false} \{b \equiv (s \wedge t) \vee (s \vee t)\}$?

Solution $b \equiv t$.

2. It is claimed that each of the following two programs stores the maximum of a , b and c in m . Prove the claim. Use true as the precondition and $m \geq a \wedge m \geq b \wedge m \geq c$ as the postcondition in both cases.

```
if  $a > b$  then  $n := a$  else  $n := b$  endif;
if  $n > c$  then  $m := n$  else  $m := c$  endif
```

And,

```
if  $a > b$  then
  if  $a > c$  then  $m := a$  else  $m := c$  endif;
else
  if  $b > c$  then  $m := b$  else  $m := c$  endif
endif
```

3. For positive integer n , prove the following program with the given specification. Use the invariant $2^{lg} = \text{pow} \leq n$.

```
{  $n > 0$  }
 $lg := 0$ ;  $pow := 1$ ;
while  $2 \times \text{pow} \leq n$  do
```

```

    lg := lg + 1; pow := 2 × pow
  enddo
  { 2lg ≤ n < 2lg+1 }

```

4. The following program divides integer variable x by a positive integer y . It stores the quotient in q and the remainder in r , both integer variables. Complete the proof.

```

  { y > 0 }
  q := 0; r := x;
  while y ≤ r do
    r := r - y; q := q + 1
  enddo
  { q × y + r = x, r < y }

```

5. The following program computes the product of x and y in z . It is a variation of the **Better multiplication** program in Figure 4.3 (page 162) in that it uses a nested loop. Complete the proof.

```

  { x × y = prod, y ≥ 0 }
  z := 0;
  while y ≠ 0 do
    while even(y) do
      x := 2 × x; y := y ÷ 2
    enddo ;
    z := z + x; y := y - 1
  enddo
  { z = prod }

```

6. In this problem, $A[0..N]$ is an array of distinct integers where N is even and $N > 0$.

Find the largest and smallest numbers and store them in hi and lo , respectively, using the following algorithm. First, compare every adjacent pair of numbers, $A[2 \times i]$ with $A[2 \times i + 1]$, for all $i, i \geq 0$, storing the smaller one in $A[2 \times i]$ and the larger one in $A[2 \times i + 1]$. Next, find the smallest number over all even-indexed elements and store it in lo , and find the largest one over all odd-indexed elements and store it in hi .

Develop the program and its proof.

7. Use the strategy given in Exercise 6 to sort $A[0..N]$, $N \geq 1$. Find the smallest and largest numbers in A and exchange them with $A[0]$ and $A[N]$, respectively.

Repeat the process with the array $A[1..N - 1]$. Write an iterative program and prove its correctness.

8. Let $A[0..N, 0..N]$, $N \geq 1$, be a matrix consisting of binary numbers where the numbers in each column are non-decreasing. The following program claims to find the first row which has all 1's. Prove its correctness.

```

i, j = 0, N - 1;
while i < N ∧ j ≥ 0 do
  if  $A[i, j] = 1$ 
    then  $j := j - 1$ 
    else  $i := i + 1$ 
  endif
enddo ;
if i < N then {j < 0}
  “row i is the desired row”
else “there is no such row”
endif

```

9. The following program computes the intersection of two non-empty finite sets, stored in arrays $A[0..M]$ and $B[0..N]$ in increasing order. The result is stored in $C[0..R]$ in increasing order where R is also returned as a result of the computation. Indices i, j and k are associated with A, B and C , respectively. Prove the correctness of the program.

```

i, j, k := 0, 0, 0;
while i < M ∧ j < N do
  if  $A[i] = B[j]$ 
    then  $C[k] := A[i]$  ;  $i, j, k := i + 1, j + 1, k + 1$ 
  else
    if  $A[i] < B[j]$ 
      then  $i := i + 1$ 
    else { $A[i] > B[j]$ }  $j := j + 1$ 
    endif
  endif
enddo ;
R := k

```

Hint For the invariant observe that $V[0..k]$ is the intersection of the scanned portions of A and B , $A[0..i]$ and $B[0..j]$. Further, $A[i]$ is larger than each element

in the scanned portion of B , and $B[j]$ is larger than each element in the scanned portion of A .

10. Consider the problem described in Section 4.7.4. Prove the same result for a matrix whose elements are real numbers. Observe that the metric used in that section, $N - s$, is now a real number, so it is not a suitable metric.

Hint Observe that the number of possible value for each matrix element x is just two, either x or $-x$. So, the number of possible values for the matrix is finite. Use a metric similar to the one used in Section 4.7.3.

11. Euclid's algorithm, given below, computes $\text{gcd}(m, n)$. Prove its correctness. Which properties of gcd are required in the proof?

```

{  $m \geq 0, n \geq 0, m \neq 0 \vee n \neq 0$  }
 $u, v := m, n;$ 
while  $v \neq 0$  do
   $q := \lfloor u/v \rfloor;$ 
   $u, v := v, u - v \times q$ 
enddo
{  $\text{gcd}(m, n) = u$  }

```

Hint Use the invariant: $u \geq 0, v \geq 0, u \neq 0 \vee v \neq 0, \text{gcd}(m, n) = \text{gcd}(u, v)$. The needed properties of gcd are: (1) $\text{gcd}(u, v) = \text{gcd}(v, u - v \times q)$, where $q = \lfloor u/v \rfloor$ is the quotient of division u by v , and (2) $\text{gcd}(u, 0) = u$.

12. Section 3.2.1.2 (page 87) has a proof of Bézout's identity that for any two positive integers m and n , there exist integers a and b such that $a \times m + b \times n = \text{gcd}(m, n)$. I develop a program starting with Euclid's algorithm for gcd given in Exercise 11 to compute a and b for any m and n (see Dijkstra [1993] for its development).

Augment the program in Exercise 11 by introducing auxiliary variables a, b, c, d , and expressions a', b', c', d' , such that $(a \times m + b \times n = u) \wedge (c \times m + d \times n = v)$ is a loop invariant. You have to derive the expressions a', b', c', d' .

```

{  $m \geq 0, n \geq 0, m \neq 0 \vee n \neq 0$  }
 $u, v := m, n; a, b := 1, 0; c, d := 0, 1;$ 
while  $v \neq 0$  do
   $q := \lfloor u/v \rfloor;$ 
   $u, v := v, u - v \times q;$ 
   $a, b, c, d := a', b', c', d'$ 
enddo

```

The postcondition, from $a \times m + b \times n = u$ and from the program in Exercise 11 that $\gcd(m, n) = u$, establishes $a \times m + b \times n = \gcd(m, n)$.

Solution Use backward substitution to derive

$$a', b', c', d' = c, d, a - c \times q, b - d \times q.$$

13. Design a proof rule for the following programming construct, “repeat”.

repeat f **until** b **endrepeat**

This command first executes f and upon termination of f tests b . If b is *false*, then repeat execution of the command, otherwise (if b is *true*) the command terminates. Note that unlike **while** b **do** f **enddo** repeat command executes f at least once.

14. Show that the following two programs are equivalent for total correctness, that is, whatever can be proved for one program (in the given proof theory) can be proved for the other.

(a) **while** b **do** f **enddo**

(b) **if** b
 then f ;
 while b **do** f **enddo**
 else *skip*
endif

15. (Bubble sort)

- (a) A very old sorting method, called *Bubble sort*, works by repeatedly picking any two items of array $A[0..N]$ that are out of order and transposing them. The algorithm terminates when there is no out of order pair. Write a program for bubble sort and prove its correctness. You may assume that transposing two items of an array permutes the array elements.

Hint Use the following non-deterministic assignment to compute array indices i and j such that $A[i]$ and $A[j]$ are out of order, that is, $i < j$ and $A[i] > A[j]$.

$$(i, j) := \{(u, v) \mid u < v, A[u] > A[v]\}$$

- (b) Consider a version of bubble sort in which only adjacent out of order pairs are transposed. Prove its correctness and show an exact bound for the number of transpositions.

Solution

- (a) Introduce a derived variable *outOrder* for the set of pairs that are out of order, $outOrder = \{(u, v) \mid u < v, A[u] > A[v]\}$. The abstract bubble sort program is:

```

{  $A_0 = A$  }
while  $outOrder \neq \{\}$  do
   $(i, j) \in outOrder$ ;
   $A[i], A[j] := A[j], A[i]$ 
enddo
{  $A$  is a permutation of  $A_0$ ,  $A$  is sorted }

```

The invariant is that A is a permutation of A_0 . On termination, $outOrder = \{\}$, so the array is sorted. To prove termination, observe that A decreases lexicographically in each iteration because a smaller element replaces a larger element toward the left-end of A . Array A has the smallest value in lexicographic ordering when it is sorted.

- (b) Transposing only adjacent items is a special case of part (a); so, the same arguments apply for its correctness. Next, show that the number of inversions decreases by 1 with each transposition, see Section 3.4.1 (page 101) for the definition of inversion. The number of inversions is at most $N^2/2$ for an array of length N , and it has this value for an array that is initially sorted in descending order. So, the number of transpositions is $\Theta(N^2)$.
16. The following algorithm is used to decide if a positive integer n is divisible by 9. Below, $sum(s)$ returns the sum of the digits of s in its decimal representation.

```

{  $n > 0$  }
 $s := n$ ;
while  $s > 9$  do
   $s := sum(s)$ 
enddo ;
 $divisible := (s = 9)$ 
{  $divisible \equiv n$  is divisible by 9 }

```

- (a) Prove the correctness of the algorithm.
- (b) Express the number of iterations as a function of n .

- (c) Given that the computation of $\text{sum}(s)$ takes $\mathcal{O}(\log s)$ time, what is the running time of the algorithm?

Hint

- (a) Show that $s \equiv^{\text{mod } 9} n$ is an invariant. Modify the solution of Exercise 38 of Chapter 3 (page 136) to prove the invariant.
- (b) If n has d digits in its decimal representation, the recurrence for the number of iterations is

$$T(d) = \begin{cases} 0 & \text{if } d = 1 \\ 1 + T(\log d) & \text{otherwise} \end{cases}$$

The solution of this recurrence relation is a function known as the iterated logarithm and written as $\log^*$. It is the number of times $\log$ has to be applied to its argument for the value to become less than or equal to 1. This is an extremely slow-growing function. Its value is at most 5 for any argument that we are ever likely to use in computing. See a longer discussion of this function in Section 6.7.2.4 (page 310).

- (c) The recurrence relation for the running time if n has d digits is:

$$R(d) = \begin{cases} \mathcal{O}(1) & \text{if } d = 1 \\ \log d + R(\log d) & \text{otherwise} \end{cases}$$

Show that $R(d) = \mathcal{O}(\log d)$.

17. Derive the exact solution of the recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot T(n-1) + 2^n & \text{if } n > 0 \end{cases}$$

Hint Verify the solution $T(n) = n \cdot 2^n$.

18. You are given a sequence of positive integers whose sum is n . Every number in the sequence is at least as large as the sum of all the numbers in the proper prefix preceding it. What is the maximum possible length of the sequence as a function of n ?

Solution Let the last number in the sequence be m , and the sum of prefix excluding the last number k . So, $n = k + m$ and $k \leq m$; so, $k \leq n/2$. Suppose the maximum possible length of the sequence is $L(n)$. Then $L(n) = (\max k : 0 \leq k \leq n/2 : 1 + L(k))$. We may assume that L is a monotonic function of n , so $L(k)$ has its maximum value at $k = n/2$. Solving $L(n) = 1 + L(n/2)$, we get $L(n) = \Theta(\log n)$.

4.A Appendix: Proof of Theorem 4.1

The statement of Theorem 4.1 (page 165) along with the accompanying conditions are as follows. First the condition of domain-closure:

$$\{b(x), x \in D\} f \{x \in D\}. \quad (\text{domain-closure})$$

Next, the specification of a loop computing a function $F : D \rightarrow D$.

$$\{x \in D, x = x_0\} \text{ while } b(x) \text{ do } f \text{ enddo } \{x = F(x_0)\}. \quad (\text{A})$$

Statement of Theorem 4.1

Assume domain-closure. Then (A) holds iff both of the following conditions hold.

1. $\neg b(x) \wedge x \in D \Rightarrow x = F(x)$,
2. $\{x \in D, b(x), F(x) = c\} f \{F(x) = c\}$, for any c in D .

Proof. Proof is by mutual implication.

- Assume domain-closure, (1) and (2). I derive (A). For all x and c :

$\{x \in D, b(x), F(x) = c\} f \{F(x) = c\}$
 , from (2)
 $\{x \in D, b(x), F(x) = c\} f \{x \in D, F(x) = c\}$
 , $x \in D$ is invariant from domain-closure
 $\{x \in D, F(x) = c\} \text{ while } b(x) \text{ do } f \text{ enddo } \{\neg b(x), x \in D, F(x) = c\}$
 , proof rule for the loop command
 $\{x \in D, F(x) = c\} \text{ while } b(x) \text{ do } f \text{ enddo } \{x = F(x), F(x) = c\}$
 , Rule of consequence on the postcondition and condition (1)
 $\{x \in D, F(x) = F(x_0)\} \text{ while } b(x) \text{ do } f \text{ enddo } \{x = F(x), F(x) = F(x_0)\}$
 , instantiate c by $F(x_0)$
 $\{x \in D, x = x_0\} \text{ while } b(x) \text{ do } f \text{ enddo } \{x = F(x_0)\}$
 , Rule of consequence on pre- and postconditions

- Assume that domain-closure and (A) hold. I derive (1) and (2).

Rewrite (A) using the equivalent way of writing a loop by unfolding one iteration, see Exercise 14 in Section 4.5.5.

$\{x \in D, x = x_0\}$
 if $b(x)$ then f ; while $b(x)$ do f enddo else skip endif
 $\{x = F(x_0)\}.$

Proof of (1): Any x for which $\neg b(x)$ holds, the given program is equivalent to *skip*. Using the proof rule for *skip*, $(\neg b(x), x \in D, x = x_0) \Rightarrow (x = F(x_0))$, that is, $(\neg b(x), x \in D, x = x_0) \Rightarrow (x = x_0, x = F(x_0))$. Eliminate $x = x_0$ from the antecedent and substitute x for x_0 in $F(x_0)$ in the consequent to get (1).

Proof of (2): Given any x for which $b(x)$ holds, the given program is equivalent to:

f ; **while** $b(x)$ **do** f **enddo**.

Annotate this expanded program and label its assertions:

$$\begin{aligned} &\{\alpha :: x \in D, b(x), x = x_0\} \\ &\quad f; \\ &\{\beta :: x \in D, x = x'\} \\ &\quad \textbf{while } b(x) \textbf{ do } f \textbf{ enddo} \\ &\{\gamma :: x = F(x_0), x = F(x')\} \end{aligned}$$

Here $\{\alpha\}f\{\beta\}$ merely asserts that after the execution of f , the value of x is still in D and it is stored in the auxiliary variable x' . Next, use (A) to justify the triple,

$\{\beta\} \textbf{while } b(x) \textbf{ do } f \textbf{ enddo} \{\gamma\}$.

From γ , $F(x_0) = F(x')$, so the value of $F(x)$ remains unchanged by the execution of f , that is, (2) holds. ■

4.B Appendix: Termination in Chameleons Problem

I presented a strategy for moving the chips in the chameleons problem in Section 4.7.2 (page 173) so that if the number of chips in two piles, say p and q , initially satisfy $p \equiv q \pmod 3$, then the game will terminate. At every point: if $p = 0$, then move chips from Q and R to P , otherwise (if $p \neq 0$) move chips from P and Q to R . I prove the correctness of this strategy in this section.

Chameleons Problem

```

while  $(p + q \neq 0) \wedge (q + r \neq 0) \wedge (r + p \neq 0)$  do
  if  $p = 0$  then { move a chip from each of  $Q$  and  $R$  to  $P$  }
     $p, q, r := p + 2, q - 1, r - 1$ 
  else { move a chip from each of  $P$  and  $Q$  to  $R$  }
     $p, q, r := p - 1, q - 1, r + 2$ 
  endif
enddo

```

Figure 4.9 Solution of the chameleons problem.

Formal description of the strategy The **Chameleons** program in Figure 4.9 (page 201) implements this strategy. The program terminates if any two piles are empty; this condition is coded as $(p + q = 0) \vee (q + r = 0) \vee (r + p = 0)$. I show that the following invariant I holds in the program:

$$I :: p \geq 0, q \geq 0, r \geq 0, p \leq q, p \equiv^{\text{mod } 3} q.$$

Proof of the invariant Below, use the abbreviation *run* for the predicate $(p + q \neq 0) \wedge (q + r \neq 0) \wedge (r + p \neq 0)$ for continuation of the loop. To prove that predicate I is invariant, show:

- (1) I holds initially,
- (2) $\{I, \text{run}, p = 0\} \ p, q, r := p + 2, q - 1, r - 1 \ \{I\}$, and
- (3) $\{I, \text{run}, p \neq 0\} \ p, q, r := p - 1, q - 1, r + 2 \ \{I\}$.

Proposition (1) holds by our assumption. Rewrite proposition (2) by expanding I ,

$$\begin{aligned} & \{p \geq 0, q \geq 0, r \geq 0, p \leq q, p \equiv^{\text{mod } 3} q, \text{run}, p = 0\} \\ & \quad p, q, r := p + 2, q - 1, r - 1 \\ & \{p \geq 0, q \geq 0, r \geq 0, p \leq q, p \equiv^{\text{mod } 3} q\} \end{aligned}$$

Using backward substitution, the verification condition is:

$$\begin{aligned} & p \geq 0, q \geq 0, r \geq 0, p \leq q, p \equiv^{\text{mod } 3} q, \text{run}, p = 0 \\ \Rightarrow & \\ & p + 2 \geq 0, q - 1 \geq 0, r - 1 \geq 0, p + 2 \leq q - 1, p + 2 \equiv^{\text{mod } 3} q - 1 \end{aligned}$$

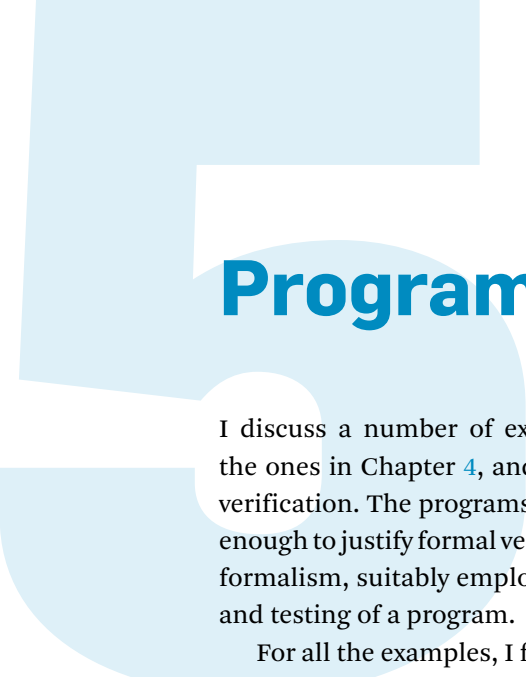
Simplifying:

$$\begin{aligned} & q \geq 0, r \geq 0, 0 \equiv^{\text{mod } 3} q, q \neq 0, r \neq 0, p = 0 \\ \Rightarrow & \\ & q - 1 \geq 0, r - 1 \geq 0, 2 \leq q - 1, 2 \equiv^{\text{mod } 3} q - 1. \end{aligned}$$

The predicates in the consequent are proved as follows: (i) from $q \geq 0, q \neq 0, 0 \equiv^{\text{mod } 3} q$, we have $q \geq 3$, so $q - 1 \geq 0$ and $2 \leq q - 1$; (ii) from $r \geq 0, r \neq 0$, we get $r - 1 \geq 0$; and (iii) from $0 \equiv^{\text{mod } 3} q$ conclude $2 \equiv^{\text{mod } 3} q - 1$. Proof of proposition (3) is similar.

Formal proof of termination Use q as the metric. This metric is well-founded because q is integer-valued and $q \geq 0$ from I . To show that q decreases in each iteration, there are two subproofs corresponding to the two branches of the conditional, each of which is easy to establish:

- (1) $\{I, \text{run}, q = q_0, p = 0\} \ p, q, r := p + 2, q - 1, r - 1 \ \{\neg \text{run} \vee q < q_0\}$,
- (2) $\{I, \text{run}, q = q_0, p \neq 0\} \ p, q, r := p - 1, q - 1, r + 2 \ \{\neg \text{run} \vee q < q_0\}$.



Program Development

I discuss a number of examples in this chapter that are more elaborate than the ones in Chapter 4, and their treatments illustrate deeper aspects of program verification. The programs treated here are not large in size, but they are complex enough to justify formal verification. The purpose of this chapter is to illustrate that formalism, suitably employed, can eliminate much speculation, hand simulation and testing of a program.

For all the examples, I first explain the problem informally. Then I describe the problem formally, develop the necessary theories and the underlying mathematics for a solution, and suggest an abstract program based on the theory. The goal of refinement is to take an abstract program and replace its fragments by program fragments that are closer to an actual implementation. A major advantage of developing a program through a series of refinements is that the proof required at every stage is limited to the portion that is being refined, and that a program fragment is replaced by another that obeys the same specification; so, the final program needs no additional proof.

For all the programs in this chapter, I suggest the invariants and variant functions, but in many cases do not show complete program annotation. I leave this, mostly mechanical, task to the reader.

5.1 Binary Search

Binary search is well-known to every beginning computer scientist. Yet writing a correct program for it is surprisingly difficult because of the corner cases. This is precisely the kind of problem where verification is most effective.

Given is an array of numbers, $A[0..N+1]$, where $N \geq 1$. According to the convention about intervals (see Section 2.1.2.5, page 9), $A[i]$ is defined for $0 \leq i \leq N$, and the array length, $N + 1$, is at least two. Recall that $i..j$ is an *interval*, whereas $A[i..j]$ is a *segment* of array A .

It is given that $A[i] \leq A[i+1]$ for all i , $0 \leq i < N$. (Even though I assume the array elements to be numbers, they could be any totally ordered data.) It is required to

find if a given number x occurs in A given that $A[0] \leq x < A[N]$. Additionally, the algorithm finds an index i such that $A[i] \leq x < A[i+1]$.

A naive procedure is to search through all items of A linearly, from index 0 to N . But using the ordered property of A , we can do better: pick any index h strictly between 0 and N ; if $A[h] \leq x$, then the desired index i is greater than or equal to h , and if $A[h] > x$, then i is less than h . Using this observation, binary search chooses h so that the segment to be searched is halved after every test; therefore, the procedure terminates in $\mathcal{O}(\log N)$ steps.

The program in Figure 5.1 (page 204) is inspired by Dijkstra [1999], who also gives a detailed derivation of it (also see Hehner [1993, p. 53]). Dijkstra establishes

$$(A[i] \leq x < A[i+1]) \wedge (\text{present} \equiv x \in A)$$

at program termination where *present* and i are boolean and integer variables, respectively; so $A[i] = x$ iff *present* is *true*. Instead of a true binary search that assigns h a value so that the segment is (nearly) halved, Dijkstra chooses the value of h using a non-deterministic assignment, $h : \in \{k \mid i < k < j\}$, where $i+1 < j$ holds as a precondition. This permits several different implementations as special cases, such as binary search, which is suitable when the array elements are uniformly distributed in value, and interpolation search, which is the way people look up a word in a physical dictionary.

Binary search

```

{ $(\forall k : 0 \leq k < N : A[k] \leq A[k+1])$ ,  $A[0] \leq x < A[N]$ }
i := 0; j := N;
while j  $\neq$  i + 1 do
  { $i+1 < j$ } h :  $\in \{k \mid i < k < j\}$  { $i < h < j$ };
  if  $A[h] \leq x$  then i := h
  else { $A[h] > x$ } j := h
endif
enddo
present := ( $A[i] = x$ )
{ $0 \leq i < N$ ,  $A[i] \leq x < A[i+1]$ , present  $\equiv$  ( $x \in A$ )}
```

Figure 5.1

The partial correctness of **Binary search** can be established using the following loop invariant. I urge the reader to complete a program annotation.

$$I:: 0 \leq i < j \leq N, A[i] \leq x < A[j].$$

Proof of termination Use the metric $j - i$. The metric value is an integer, and given $0 \leq i < j \leq N$ as part of the loop invariant, it is non-negative (actually, positive). So, it is well-founded. Next, we need to show that

$$\{I \wedge j \neq i + 1 \wedge j - i = m\} \text{ loop body } \{j - i < m\}.$$

This involves proving the following two propositions corresponding to the two branches of the conditional; use the fact that $i < h < j$ is a postcondition of $h := k \mid i < k < j$.

1. $\{I \wedge j \neq i + 1 \wedge j - i = m \wedge i < h < j \wedge A[h] \leq x\} \quad i := h \quad \{j - i < m\}.$

This follows simply from

$$\{j - i = m \wedge i < h < j\} \quad i := h \quad \{j - i < m\}.$$

$$\text{That is, } j - i = m \wedge i < h < j \Rightarrow j - h < m.$$

2. $\{I \wedge j \neq i + 1 \wedge j - i = m \wedge i < h < j \wedge A[h] > x\} \quad j := h \quad \{j - i < m\}.$

This follows simply from

$$\{j - i = m \wedge i < h < j\} \quad j := h \quad \{j - i < m\}.$$

$$\text{That is, } j - i = m \wedge i < h < j \Rightarrow h - i < m.$$

The execution time of **Binary search** yields the recurrence relation:

$$T(n) = \max(T(h), T(n - h)) + c$$

where the first comparison is made with the element at position h , $h > 0$. If h is small, say 1 for each iteration so that at most one element is eliminated each time, the running time is $\mathcal{O}(n)$. However, with h approximately equal to $n/2$, the recurrence is $T(n) = T(n/2) + c$ whose solution is $T(n) = \Theta(\log n)$. Correctness is independent of the choice of h , but the running time is strongly dependent on it.

5.2 Saddleback Search

A method to locate a given value in a matrix that is sorted by rows and by columns has been named *Saddleback search* by David Gries. Given is a matrix of numbers $V[0..R + 1, 0..C + 1]$, where the rows are numbered 0 through R from the top to bottom and the columns 0 through C from left to right. Each row is sorted in ascending order from left to right and each column from top to bottom, that is, $V[i, j] \leq V[i, j + 1]$ and $V[i, j] \leq V[i + 1, j]$, for the proper values of the indices. Given that value x occurs in V , find its position, that is, determine i and j such that $V[i, j] = x$.

It may seem that some form of binary search along the rows and/or columns would be appropriate to eliminate some submatrix (analogous to eliminating an array segment in binary search). Actually, saddleback search proceeds to eliminate a single row or a column in each step, thus requiring $\mathcal{O}(R + C)$ steps. Start the search

from the upper right corner of the matrix, comparing x with $V[0, C]$. If $x < V[0, C]$, then x is smaller than all elements of column C , so eliminate that column. And, if $x > V[0, C]$, then x is larger than all elements of row 0, so eliminate that row. And if $x = V[0, C]$, then we have found its position. The search is always limited to the matrix whose bottom left corner has the fixed indices $(R, 0)$ and upper right corner the variable indices (i, j) .

Define predicate $2sorted(i, j)$ to denote that the submatrix described above is sorted by both rows and columns, that is,

$(\forall m, n : m \in 0..i, n \in 0..j : V[m, n] \leq V[m, n+1] \wedge V[m, n] \leq V[m+1, n])$, and predicate $in(i, j)$ to denote that x is in the given submatrix, that is,

$(\exists m, n : m \in 0..i, n \in 0..j : x = V[m, n])$.

The program in Figure 5.2 needs no further explanation.

Saddleback search

```

{2sorted(0, C), in(0, C)}
  i, j := 0, C;
  while x ≠ V[i, j] do
    if x < V[i, j]
      then {x < V[i, j]}   j := j - 1
      else {x > V[i, j]}   i := i + 1
    endif
  enddo
  {V[i, j] = x}

```

Figure 5.2

The reader is urged to complete the annotation of **Saddleback search**. Use the loop invariant:

$(0 \leq i \leq R, 0 \leq j \leq C, in(i, j), 2sorted(i, j))$.

For the termination argument, the obvious metric is the size of the matrix under consideration. In fact, a much simpler metric, $j - i$, suffices. The smallest value of $j - i$ is for $j = 0$ and $i = R$, that is, $-R$, and the largest value is C . Therefore, $j - i$ takes values from a well-founded set spanning $-R$ through C . Each iteration decreases $j - i$ by either decreasing j or increasing i , so the program terminates.

The proof of termination also gives the best estimate of the running time; it is $\mathcal{O}(R + C)$. This is not very good if the R and C are vastly different in magnitude. In

particular, for $R = 1$ the saddleback search degenerates to a linear search starting at the right end of the row, a situation where binary search can be applied. Exercise 6 (page 257) asks you to adapt binary search for situations where R and C are very different.

Observe that binary search finds the roots of a discrete one-dimensional monotonic function over a finite interval, whereas saddleback search finds the roots of a discrete two-dimensional monotonic function over finite intervals.

5.3 Dijkstra's Proof of the am-gm Inequality

Dijkstra [1996a] proposes a novel proof of the am-gm inequality, that the arithmetic mean of a bag of positive reals is at least the geometric mean, based on the ideas of program proving; see Section 3.4.2 (page 102) for inductive proofs of this inequality. In particular, Section 3.4.2.2 (page 104) has a proof using problem reformulation that is somewhat simpler than Dijkstra's proof.

Denote the arithmetic mean and the geometric mean of the given bag B by a_0 and g_0 , respectively. As long as all items of B are not equal, transform B such that: (1) its arithmetic mean does not increase, (2) the geometric mean remains the same and (3) the number of items that differ from the geometric mean decreases.

Formalize the construction by a program in which a is the arithmetic mean, g the geometric mean and ndg the number of elements of B that differ from g . The partially annotated program skeleton is given below, where a_0, g_0 and k are auxiliary variables:

```

{  $a = a_0, g = g_0$  }
while  $ndg \neq 0$  do
{  $a_0 \geq a, g = g_0, ndg = k$  }
   $f$ 
{  $a_0 \geq a, g = g_0, ndg < k$  }
enddo
{  $a_0 \geq g_0$  }

```

I now complete the proof of the am-gm inequality.

- Using ndg as a metric, the loop terminates. Suppose f meets its specification, that is,

$$\{a_0 \geq a, g = g_0, ndg = k\} \quad f \quad \{a_0 \geq a, g = g_0, ndg < k\}.$$

The annotation then shows that $a_0 \geq a, g = g_0$ is an invariant of the loop. On termination:

$$\begin{aligned}
& a_0 \geq a, g = g_0, ndg = 0 \\
\Rightarrow & \{ndg = 0 \Rightarrow \{\text{all items of } B \text{ are equal}\} a = g\}
\end{aligned}$$

$$\begin{aligned}
& a_0 \geq a = g = g_0 \\
\Rightarrow & \{\text{arithmetic}\} \\
& a_0 \geq g_0
\end{aligned}$$

• Dijkstra proposes an f that meets the given specification. Let x be a smallest element and y a largest element in B at any point. Given that $ndg \neq 0$, that is, not all elements of B are equal, $x < g < y$. Let f transform B by replacing the pair (x, y) by $(g, x \cdot y/g)$. I show that f meets its specification.

$$\begin{aligned}
& x < g < y \\
\Rightarrow & \{\text{arithmetic}\} \\
& (g - x) \cdot (g - y) < 0 \\
\Rightarrow & \{\text{expand and rearrange terms}\} \\
& (g^2 + x \cdot y) < (g \cdot (x + y)) \\
\Rightarrow & \{g > 0. \text{ Divide both sides by } g\} \\
& (g + x \cdot y/g) \leq (x + y)
\end{aligned}$$

Therefore, as result of executing f : (1) a does not increase because $x + y$ in the sum of the elements of B is replaced by $g + x \cdot y/g$, (2) g does not change because $x \cdot y = g \times x \cdot y/g$ and (3) the number of elements of B equal to g increases by at least 1, so ndg decreases.

5.4 Quicksort

A classic algorithm for sorting elements of an array in place, known as *quicksort*, due to Hoare [1961], is presented and proved in this section. The algorithm is inherently recursive. I express parts of the algorithm recursively and the other parts, where the array elements have to be permuted, imperatively. The proof rules for recursive and non-recursive procedures are identical; so this poses no difficulty.

There have been a number of enhancements of *quicksort* since the publication of Hoare [1961]; see in particular Bentley and McIlroy [1993] that carefully measures the running time for a number of different heuristics. The goal of this section is not to discuss the most efficient *quicksort* but explain proof methods for recursive programs using *quicksort* as an example.

Sorting is typically applied to records where each record has a key and, possibly, other data. Records have to be sorted based on the keys, and the most we can do is to compare different keys and move the records around. It is not possible to apply arithmetic operations on keys, for instance. In particular, the keys may be just 0 and 1 so that the records with 0-key precede the records with 1-key after sorting.

Algorithm outline The *quicksort* algorithm, in broad terms, is as follows. First, choose an element p of the array as a *pivot* element. Then call *partition* to permute the array elements so that they form three segments, *left*, *center* and *right*, where (1) the center segment has just the pivot element p , (2) every element of the left segment is less than or equal to p and (3) every element of the right segment is greater than p . The partitioning is shown schematically in Table 5.1 where L, C and R denote the left, center and right segments, respectively. Either or both of L and R may be empty. This partitioning ensures that elements across different segments are in order, though the elements within a segment may be out of order.

Table 5.1 Partitioning in *quicksort*

L	C	R
$\leq p$	$= p$	$> p$

Next, apply *quicksort* to the left and right segments independently. The correctness of the scheme is easy to see: inductively, the left and right segments are sorted, so elements within each segment are in order. And partitioning has guaranteed that the elements across different segments are in order.

The algorithm can be described succinctly in a concurrent programming language where both partitioning of an array segment and sorting of different segments can all proceed concurrently [see [Kitchin et al. 2010](#)].

Terminology and notations Henceforth, $Q[0..N]$ is a non-empty array that is to be sorted in place in ascending order. Recall the conventions about intervals from Section 2.1.2.5 (page 9), in particular that interval $u..v$ includes the left bound but excludes the right one, so its length is $v - u$ if $u \leq v$. Interval $u..v$ is empty if $u \geq v$. Interval $s..t$ is a sub-interval of $u..v$ if $u \leq s \leq t \leq v$. A *segment* $Q[u..v]$ is the sequence of consecutive array elements corresponding to interval $u..v$. The entire array Q is $Q[0..N]$.

Write $\text{sorted}(u..v)$ to denote that $Q[u..v]$ is sorted in ascending order and $Q[u..v] \leq p$ to denote that every element of the given segment is smaller than or equal to p ; analogously $Q[u..v] > p$. Some associated facts that we use are: (1) for empty interval $u..v$, both $Q[u..v] \leq p$ and $Q[u..v] > p$ are *true* for any p , and (2) given $Q[u..v] \leq p$ (or $Q[u..v] > p$) and that $s..t$ is a sub-interval of $u..v$, deduce $Q[s..t] \leq p$ (or $Q[s..t] > p$).

5.4.1 Specification of *quicksort*

A call to procedure *quicksort*(u, v) sorts the segment $Q[u..v]$ without accessing other parts of Q . Array Q is a global variable, not a parameter in the call. The value parameters u and v are not changed in *quicksort*, which I do not explicitly show in

the specification. Below, the specification says that *quicksort*(u, v) sorts the segment $Q[u..v]$:

$$\begin{array}{l} \{u \leq v\} \\ \text{quicksort}(u, v) \\ \{\text{sorted}(u..v)\} \end{array}$$

This specification is incomplete because it does not specify any relationship between the array elements before and after application of the procedure. So, an implementation of *quicksort* can merely fill the segment with natural numbers in order, thus meeting the specification. Tighten the specification by requiring in the postcondition that Q is a permutation of the original array Q_0 . This specification is still deficient because it permits the segment $u..v$ to contain elements that were originally outside the segment. Therefore, further require that the permutation be restricted to the elements in segment $Q[u..v]$.

Write $Q \stackrel{u..v}{=} Q_0$ to denote that Q and Q_0 are permutations of each other within the interval $u..v$ and identical outside $u..v$. The specification now reads:

$$\begin{array}{l} \{u \leq v, Q = Q_0\} \\ \text{quicksort}(u, v) \\ \{\text{sorted}(u..v), Q \stackrel{u..v}{=} Q_0\} \end{array}$$

Derive from the postcondition of *quicksort*($0, N$) that Q is sorted and Q is a permutation of its initial values:

$$\begin{array}{l} \{0 \leq N, Q = Q_0\} \\ \text{quicksort}(0, N) \\ \{\text{sorted}(0..N), Q \stackrel{0..N}{=} Q_0\} \end{array}$$

Structure of the proof The code of *quicksort*(u, v) is recursive. The proof of *quicksort*(u, v), therefore, uses the specification of *quicksort* for the recursive calls. This scheme emphasizes that the code of a called procedure, say P' , is unnecessary for a proof of its caller, P ; all that is required to prove P is a specification of P' .

Procedure *quicksort* modifies Q by only exchanging pairs of its elements; so, it creates a permutation. Further, any procedure that has an interval as its parameter permutes only the corresponding segment. This fact can be proved at each step in the annotated proof, but I do not show this explicitly.

I use without proof that for every interval $u..v$ the end points u and v are within 0 and N , that is, $0 \leq u < N$ and $0 \leq v < N$. Other parts of the proof are given in detail, but the verification conditions are not proved; they can mostly be proved by inspection.

5.4.2 Procedure *quicksort*: Implementation and Proof

The annotated code of *quicksort* is shown in Figure 5.3 (page 211). The annotation uses the specification of *partition* that appears below Figure 5.3, which is proved in Section 5.4.3 (page 212).

Annotated code of *quicksort*

```

{u ≤ v}
Proc quicksort(u, v)
  if u + 1 < v { the segment has more than one element }
  then { u + 1 < v }
    partition(u, v, s);
    {u ≤ s < v, Q[u..s] ≤ Q[s] < Q[s + 1..v]}
    quicksort(u, s);
    {u ≤ s < v, sorted(u..s), Q[u..s] ≤ Q[s] < Q[s + 1..v]}
    quicksort(s + 1, v)
    {u ≤ s < v, sorted(u..s), sorted(s + 1..v)
     , Q[u..s] ≤ Q[s] < Q[s + 1..v]}
    {sorted(u..v)}
  else { the segment has at most one element, so it is sorted. }
    {u ≤ v, u + 1 ≥ v, sorted(u..v)}
    skip
    {sorted(u..v)}
  endif
  {sorted(u..v)}
endProc
{sorted(u..v)}

```

Figure 5.3

Specification of *partition* Procedure call *partition*(*u*, *v*, *s*) has value parameters *u* and *v*, where *Q*[*u*..*v*] has at least two elements. Variable parameter *s*, returned by the procedure, separates the left and the right segments.

```

{u + 1 < v}
partition(u, v, s)
{u ≤ s < v, Q[u..s] ≤ Q[s] < Q[s + 1..v]}

```

Proof of termination of *quicksort* It is required to postulate a metric using the parameters of each recursive procedure so that any recursive call to a procedure has a lower metric value than the one associated with the caller. For *quicksort*, every recursive call has an interval as its parameter, and the size of the interval decreases

with each call. So, termination is guaranteed for *quicksort*. Proof of termination of *partition* is deferred to Section 5.4.3.

5.4.3 Procedure *partition*: Implementation and Proof

The implementation of *partition* with parameters (u, v, s) uses $Q[u]$ as a pivot element to separate the left and right segments and store the initial value of $Q[u]$ in $Q[s]$; see Figure 5.4. The first part of the program does not alter $Q[u]$ but permutes $Q[u+1..v]$ so that $Q[u+1..s+1] \leq Q[u] < Q[s+1..v]$. In particular, $Q[s] \leq Q[u]$. The next part exchanges $Q[s]$ and $Q[u]$ so that $Q[u..s] \leq Q[s] < Q[s+1..v]$, as required.

The strategy for the first part is to scan the given segment in order of increasing indices starting at $u+1$ until an element larger than $Q[u]$ is found. Next, scan the segment in order of decreasing indices from $v-1$ until an element smaller than $Q[u]$ is found. Then exchange the two elements and continue the process until there are no more elements to scan. There are a number of corner cases that need to be considered, such as that all elements could be smaller or larger than $Q[u]$, and derive the condition under which the scanning should terminate. These concerns demand a careful analysis of the program.

In the annotated code, s is used as the index for scanning in increasing order and t for scanning in decreasing order. The loop invariant is:

$$u \leq s < t \leq v, Q[u..s+1] \leq Q[u] < Q[t..v].$$

The verification conditions corresponding to the annotations in Figure 5.4 are easily proved by applying backward substitution, elementary integer arithmetic and propositional logic.

Proof of termination Use the metric $t - s$ for proof of termination of the loop. From the invariant we have $s < t$, so $t - s > 0$. Each iteration of the loop either increases s , decreases t or does both, thus decreasing $t - s$.

5.4.4 Cost of *quicksort*

Running time Execution time of *quicksort* is heavily dependent on the result of *partition*. Suppose *partition* is so perfect that it always divides the segment into nearly equal halves. Then we get the recurrence $T(n) = 2T(n/2) + \Theta(n)$ whose solution is $\Theta(n \log n)$. Actually, even if the halves are not nearly equal but each is no smaller than n/p for some fixed p , we still get $\Theta(n \log n)$ running time. In the worst case, however, *partition* may create completely unbalanced subsegments. The recurrence in that case is $T(n) = T(p) + T(n-p) + \mathcal{O}(n)$, where p is a small value independent of n . The solution to the recurrence is $\mathcal{O}(n^2)$. Ironically, this situation arises if the array is already sorted in descending order.

Annotated code of *partition*

```

Proc partition( $u, v, s$ )
 $\{u + 1 < v\}$ 
   $s, t := u, v$ ;
 $\{u \leq s < t \leq v, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
  while  $s + 1 \neq t$  do
 $\{u \leq s < t \leq v, s + 1 \neq t, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
    if  $Q[s+1] \leq Q[u]$  then
 $\{u \leq s < t \leq v, s + 1 \neq t, Q[s+1] \leq Q[u], Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
       $s := s + 1$ 
 $\{u \leq s < t \leq v, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
    else
 $\{u \leq s < t \leq v, s + 1 \neq t, Q[s+1] > Q[u], Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
      if  $Q[t-1] > Q[u]$  then
 $\{u \leq s < t \leq v, s + 1 \neq t, Q[s+1] > Q[u], Q[t-1] > Q[u],$ 
 $Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
         $t := t - 1$ 
 $\{u \leq s < t \leq v, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
      else
 $\{u \leq s < t \leq v, s + 1 \neq t, Q[s+1] > Q[u], Q[t-1] \leq Q[u],$ 
 $Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
         $Q[s+1], Q[t-1], s, t := Q[t-1], Q[s+1], s+1, t-1$ 
 $\{u \leq s < t \leq v, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
      endif
 $\{u \leq s < t \leq v, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
    endif
 $\{u \leq s < t \leq v, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
  enddo ;
 $\{u \leq s < t \leq v, s + 1 = t, Q[u..s+1] \leq Q[u] < Q[t..v]\}$ 
 $\{u \leq s < v, Q[u..s+1] \leq Q[u] < Q[s+1..v]\}$ 
 $Q[u], Q[s] := Q[s], Q[u]$ 
 $\{u \leq s < v, Q[u..s] \leq Q[s] < Q[s+1..v]\}$ 
endProc

```

Figure 5.4

It is more subtle to compute the expected running time. A simplifying assumption that is sometimes made is that the input consists of distinct items, and all their permutations are equally likely as input. Under these assumptions, the expected running time is $\Theta(n \log n)$. However, this is a grossly simplistic assumption. In reality, input items are almost never independently distributed, and often there are identical items in the input. An array of equal items, which is sorted by definition, forces the given quicksort program to its worst-case behavior, $\Theta(n^2)$.

An elegant way to handle equal items is proposed in Bentley and McIlroy [1993]. Their solution partitions a segment into five parts in order from left to right, as shown schematically in Table 5.2. Segments A and E both contain items equal to the pivot, B and D contain smaller and larger items than the pivot, respectively, and the center segment C is unscanned. Some of these segments may be empty. The partitioning strategy is to shrink C one item at a time until it becomes empty. Start scanning C from its left end. If the scanned item x is equal to the pivot, swap it with the first item of B, thus extending both A and B to the right by one position; the swap increases the length of A while that of B remains the same. If x is smaller than the pivot, extend segment B. In both cases, C shrinks by one item. If x is larger than the pivot, apply similar steps at the right end of C until an item y smaller than the pivot is found. Then swap x and y , as in the original *quicksort* partitioning. In case A or E is empty, the procedure is slightly modified. When C becomes empty, parts of A and B are swapped so that the data in B sits at the left end. By similar swapping, data in D sits at the right. Then data items whose keys match the pivot sit in between. Bentley and McIlroy report significant performance gains using 5-way partitioning.

Table 5.2 Bentley–McIlroy partition in *quicksort*

A	B	C	D	E
=	<	?	>	=

In order to meet the assumption of random input, a program may first randomize the order of the array elements. There is a simple $\mathcal{O}(n)$ randomizing algorithm, assuming that there is a random number generator that returns a number between 1 and n in constant time for any positive n . Then the pivot element is chosen randomly, and assuming that the elements are distinct, the segments after partitioning are random. The expected running time of quicksort in this case is $\Theta(n \log n)$.

Space usage There is a subtle point about space usage in *quicksort*. Recursive calls are typically implemented using a call stack in which the parameters and return addresses are saved for later execution. Let us compute the maximum stack size. A sketch of the implementation is as follows. Initially, the pair $(0, N)$ is placed on

the stack, corresponding to the call *quicksort*(0, *N*). In each iteration: if the stack is non-empty, the top element of the stack (*u*, *v*) is removed and *quicksort*(*u*, *v*) is executed. After partitioning the segment *u*..*v*, pairs (*u*, *s*) and (*s* + 1, *v*) are placed on the stack in some order, corresponding to the recursive calls, *quicksort*(*u*, *s*) and *quicksort*(*s* + 1, *v*); see **Annotated code of quicksort** in Figure 5.3 (page 211). The iterations are repeated until the stack is empty.

Henceforth, refer to a pair in the stack as a segment to simplify this presentation. Observe that the segments (*u*, *s*) and (*s* + 1, *v*) may be placed on the stack in either order without affecting correctness since they operate on disjoint segments. The choice of order is important in limiting the stack size. If the longer segment is placed on top of the shorter one, the stack size may be as large as $\mathcal{O}(N)$, the length of the array. However, placing the shorter segment on top of the longer one, that is, sorting the shorter segment first, limits the stack size to $\mathcal{O}(\log N)$ (see the solution to Exercise 18 of Chapter 4, page 199).

Implemented versions of quicksort At the time Hoare wrote the original paper on quicksort, computers did not have multi-level memory. Today a naive implementation of quicksort, as I have sketched in this section, will incur a heavy penalty in performance because the memory transfer time is substantial compared to working with a cache item. Partitioning with a single pivot element was a sensible approach then because multi-way partitioning would have been expensive with earlier machines. Modern computers are more efficient at multi-way partitioning. Current implementation in Java-7, called *dual-pivot* quicksort, uses two pivot elements to partition a segment into three parts.

5.5 Heapsort

A class of sorting algorithms, known as *selection sort*, operate as follows. The array to be sorted is partitioned into a left and a right segment, *L* and *R*, so that *R* is sorted and every element in *L* is less than or equal to every element in *R*. Initially, *L* is the entire array and *R* is empty. The computation strategy is: in each step exchange the rightmost element of *L* with its maximum element. This, effectively, extends *R* to the left by one element (recall that the maximum element of *L* is no more than any element of *R*). Since each step extends *R*, eventually the entire array becomes *R* and is sorted.

Heapsort, a selection sort algorithm, was proposed in Williams [1964] and enhanced in Floyd [1964]. The algorithm uses a clever data structure, *heap*, for *L* so that its first element is its maximum element. The algorithm outline is: (1) initially, transform the entire array into a heap *L* so that *R* is empty (therefore sorted), and (2) as long as *L* has more than one element, in each step exchange the first and the last element of *L*, thereby extending *R* to its left and shortening *L*; this exchange may destroy the heap structure, so remake *L* into a heap.

For an array of length N , the initialization step takes $\Theta(N)$ time and each step in (2) takes $\Theta(\log N)$ time. Therefore, the worst-case complexity of the algorithm is $\Theta(N \log N)$. It can be shown that the expected complexity of the algorithm is also $\Theta(N \log N)$. Both steps, (1) and (2), take constant amount of additional space.

Terminology and notations The array to be sorted is $H[1..N]$, where $N > 1$, so the array length is at least 1. (Observe that indices start at 1 instead of 0, unlike in *quicksort*.) Define *interval* $u..v$ and the corresponding *segment* $H[u..v]$ as in *quicksort*, Section 5.4. Write $H[u..v] \leq H[s..t]$ if every element in $H[u..v]$ is at most every element in $H[s..t]$. Henceforth, *sorted*($u..v$) denotes that the corresponding segment of H is sorted and *heap*($s..t$) that the segment is a heap.

5.5.1 Outline of the Main Program

I define the heap data structure in the next section. For the proof of the main program, assume that (1) every sub-segment of a heap is a heap (so, an empty segment is a heap), and (2) $H[1]$ is a maximum element of a non-empty heap $H[1..v]$. Using index j to separate L and R , the invariant is:

$$(1 \leq j \leq N, \text{heap}(1..j), \text{sorted}(j..N), H[1..j] \leq H[j..N]).$$

Specification of procedure *heapify* shows that it transforms H to a heap and that *extendHeap* extends a given heap one place to its left. Both procedures are developed subsequently.

$\begin{array}{l} \{N > 1\} \\ \text{heapify} \\ \{N > 1, \text{heap}(1..N)\} \end{array}$	$\begin{array}{l} \{u \geq 1, \text{heap}(u + 1..v)\} \\ \text{extendHeap}(u, v) \\ \{\text{heap}(u..v)\} \end{array}$
--	--

Using these two procedures, I describe the annotated code of heapsort in Figure 5.5 (page 217). The verification conditions are mostly easy to see given the two properties of heap and the formal specifications of *heapify* and *extendHeap*. The postcondition in the annotation follows from: (1) $j = 2$ and *sorted*($j..N$) implies *sorted*($2..N$), and (2) $H[1..j] \leq H[j..N]$ implies that the first element is a smallest element. So, *sorted*($1..N$) holds.

5.5.2 Heap Data Structure

The key to the efficiency of heapsort is the heap data structure. Formally, $H[u..v]$ is a heap if for every i , $u \leq i < v$, $H[i] \geq H[2 \cdot i]$ and $H[i] \geq H[2 \cdot i + 1]$, provided these indices are within the interval. It is easier to visualize a heap as a tree where each node is a pair $(i, H[i])$, with index i and value $H[i]$. The children of this node are $(2 \cdot i, H[2 \cdot i])$ and $(2 \cdot i + 1, H[2 \cdot i + 1])$ whichever of these indices are within the interval, and every element's value is at least as large as all of its children. Heap $H[1..v]$ has root with index 1; in general, heap $H[u..v]$, where $u > 1$, does not have a

Annotated code of heapsort

```

{N > 1}
Proc heapsort
  heapify;
  {N > 1, heap(1..N), sorted(N..N),  $H[1..N] \leq H[N..N]$ }
   $j := N$ ;
  {j > 1, heap(1..j), sorted(j..N),  $H[1..j] \leq H[j..N]$ }
  while j > 2 do
    {j > 2, heap(1..j), sorted(j..N),  $H[1..j] \leq H[j..N]$ }
     $H[1], H[j-1] := H[j-1], H[1]$ ;
    {j > 2, heap(2..j-1), sorted(j-1..N),  $H[1..j-1] \leq H[j-1..N]$ }
    extendHeap(1, j-1);
    {j > 2, heap(1..j-1), sorted(j-1..N),  $H[1..j-1] \leq H[j-1..N]$ }
     $j := j - 1$ 
  {j > 1, heap(1..j), sorted(j..N),  $H[1..j] \leq H[j..N]$ }
  enddo
  {j = 2, heap(1..j), sorted(j..N),  $H[1..j] \leq H[j..N]$ }
endProc
{sorted(1..N)}

```

Figure 5.5

single root. Figure 5.6 shows a rooted heap where “index: value” is written next to each node.

It follows that the value at the root of a subtree in a heap is at least as large as all its descendants. So, $H[1]$ is a maximum value in the heap $H[1..v]$, for any v ,

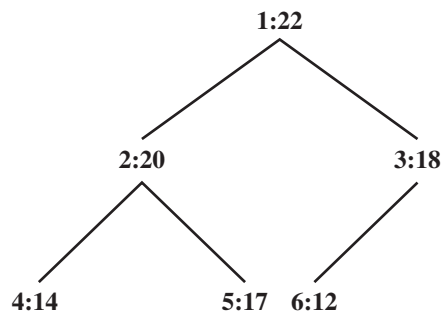


Figure 5.6 Example of a heap.

$v > 1$. Also, every sub-segment of a heap is a heap, by definition. These are the two properties claimed for a heap in Section 5.5.1 (page 216).

Define $child(i, v)$ and $desc(i, v)$, respectively, as the set of indices of children and descendants of i , $1 \leq i < v$.

$$\begin{aligned} child(i, v) &= \{j \mid j \in \{2 \times i, 2 \times i + 1\}, j < v\} \\ desc(i, v) &= child(i, v) \cup (\cup j : j \in child(i, v) : desc(j, v)) \end{aligned}$$

The definition of $desc$ is well-formed because $desc(i, v)$ is empty if $child(i, v)$ is empty.

Use $H[i] \geq H[child(i, v)]$ and $H[i] \geq H[desc(i, v)]$ with obvious meanings. Then define the “heap condition”, (HC):

$$heap(u..v) = (\forall j : u \leq j < v : H[j] \geq H[desc(j, v)]) \quad (HC)$$

5.5.3 extendHeap

Recall that $extendHeap(u, v)$ establishes $heap(u..v)$ given $heap(u + 1..v)$. I will ensure that during execution of the procedure every node in $u..v$ satisfies (HC) with one possible exception; call the node that fails to satisfy (HC), if it exists, the “misfit”. For example, if the tree in Figure 5.6 (page 217) had value 10 at the root, with index 1, that would be a misfit. Then exchange its value, 10, with its larger child’s value, 20 at node 2. This would remove the misfit at node 1 but create a misfit at node 2. Repeating the step at node 2 exchanges its value with its larger child’s value, 17 at node 5. Now node 5, which has value 10, is not a misfit because it has no children. In general, the misfit would trickle down the tree along a path in its subtree until it is not a misfit because it is larger than all its children (possibly, it has no children).

Given $heap(u + 1..v)$ if $H[u]$ satisfies (HC), then $H[u..v]$ a heap. Otherwise, the misfit in $H[u..v]$ is u and it has a child t , $t \in child(u, v)$, such that $H[u] < H[t]$; let t be the child with the maximum value. Exchange $H[u]$ and $H[t]$. Now (HC) holds at u though it may no longer hold at t . Next, repeat the procedure with node t as the misfit. Thus, each iteration either creates a heap in $H[u..v]$ or makes a child of the misfit the misfit. The latter possibility cannot continue indefinitely; so, the procedure terminates.

In the code of $extendHeap$ in Figure 5.7 (page 219), c is the possible misfit. Initially, $c = u$; finally, c is larger than all its children, possibly it has no children. The loop invariant says that $heap(u..v)$ holds for every node in $u..v$ except possibly c :

$$I :: (\forall j : u \leq j < v, j \neq c : heap(u..v)).$$

The annotation in Figure 5.7 is partly informal but formal where it is most needed, particularly in the portion between the labels *A* and *E*, to show that exchanging $H[u]$ and $H[t]$ preserves the loop invariant *I*.

Annotated code of *extendHeap*

```

{heap(u + 1..v)}
Proc extendHeap(u, v)
  c := u;
  { I :: (∀ j : u ≤ j < v, j ≠ c : heap(u..v)) }
  while 2 × c < v do
    { c has a child }
    if  $H[c] \geq H[\text{child}(c, v)]$ 
      then { no child has a larger value } skip { the subtree at c is a heap } {I}
    else { there is a larger child. Compute t, the index of the largest child. }
      if 2 × c + 1 < v ∧  $H[2 \times c + 1] > H[2 \times c]$ 
        then { there are two children and  $H[2 \times c + 1]$  is strictly larger }
          t := 2 × c + 1
        else {  $H[2 \times c]$  is the only child or it is not the smaller child }
          t := 2 × c
        endif ;
      • A: {I, t ∈ child(c, v),  $H[t] > H[c]$ ,  $H[t] \geq H[\text{child}(c, v)]$ }
      • B: { (∀ j : u ≤ j < v, j ≠ t, j ≠ c : heap(j..v)) ∧  $H[t] \geq H[\text{desc}(c, v)]$  } ∧  $H[t] \geq H[c]$  }
           $H[c], H[t] := H[t], H[c]$ ;
      • C: { (∀ j : u ≤ j < v, j ≠ t, j ≠ c : heap(j..v)) ∧  $H[c] \geq H[\text{desc}(c, v)]$  } ∧  $H[c] \geq H[t]$  }
      • D: { (∀ j : u ≤ j < v, j ≠ t : heap(j..v)) }
          c := t
      • E: {I :: (∀ j : u ≤ j < v, j ≠ c : heap(j..v)) }
    endif
    {I}
  enddo
  {2 × c ≥ v, I}
endProc
{heap(u..v)}

```

Figure 5.7

5.5.3.1 Analysis of *extendHeap*

Verification conditions The assertions up to and including the one at *A* can be verified by inspection. I prove the assertions from *B* to *E* by backward substitution. First, applying backward substitution to *E*, *D* is easily shown. Now, $C \Rightarrow D$. Again, *B* follows by applying backward substitution to *C*. And $A \Rightarrow B$.

Minor optimizations A small optimization is to terminate the loop if $H[c] \geq H[\text{child}(c, v)]$ because misfit *c* has no larger child, so $H[u..v]$ is a heap. Computation of the largest child of $H[c]$ can also be slightly improved. The assignment $H[c], H[t] := H[t], H[c]$ may be replaced by $c, H[c], H[t] := t, H[t], H[c]$.

5.5.4 *heapify*

Procedure *heapify*, proposed in Floyd [1964], constructs a heap in situ from an arbitrary array of numbers $H[1..N]$ in linear time. First observe that $H[\lceil N/2 \rceil..N]$ is a heap since no element in this segment has a child. So, in order to heapify array $H[1..N]$, start with $j = \lceil N/2 \rceil$, and repeatedly extend the heap to its left by applying *extendHeap*(*j*, *N*). The annotation in Figure 5.8 (page 220) needs little explanation.

Annotated code of *heapify*

```

{N > 1}
Proc heapify
  j := ⌈N/2⌉;
  {j ≥ 1, heap(j..N)}
  while j ≠ 1 do
    {j > 1, heap(j..N)}
    extendHeap(j - 1, N);
    {j > 1, heap(j - 1..N)}
    j := j - 1
  enddo
  {j = 1, heap(j..N)}
endProc
{heap(1..N)}

```

Figure 5.8

5.5.5 Running Time of Heapsort

There are two separate costs in heapsort, given in Figure 5.5 (page 217), for one call to *heapify* and around *N* calls to *extendHeap*(1, *j* - 1), plus a linear amount

of time for the executions of all $H[1], H[j-1] := H[j-1], H[1]$ and updating the index j .

First, estimate the running time of $extendHeap(u, v)$ as a function of the length of segment $u..v$. In each iteration, either the iteration is skipped because the subtree at c is already a heap or c is assigned the value of t , which is either $2 \times c + 1$ or $2 \times c$. Since c 's value at least doubles in each iteration, the number of iterations in any execution of $extendHeap$ is at most $\log N$, for a total cost of $\mathcal{O}(N \log N)$ over the execution of the entire algorithm.

Procedure $heapify$ calls $extendHeap$ N times. So, the cost of $heapify$ is $\mathcal{O}(N \log N)$. I show a better estimate, that the cost of $heapify$ is $\mathcal{O}(N)$. A heap is almost a full binary tree in its structure, so its maximum height h is about $\log N$ and the number of nodes at height i is approximately 2^{h-i} , $0 \leq i \leq h$. It takes i steps to execute $extendHeap(u, v)$ where the height of u is i , from the discussion in the previous paragraph. So, the total cost of $heapify$ is the sum of heights in a heap, $(+i : 0 \leq i \leq h : i \cdot 2^{h-i}) = 2^h \times (+i : 0 \leq i \leq h : i/2^i)$. Estimate this sum as follows.

Write S for $(+i : 0 \leq i : i/2^i)$. Observe that the $(i+1)^{\text{th}}$ term of S is $(i+1)/2^{i+1}$ and the i^{th} term of $S/2$ is $i/2^{i+1}$; so their difference is $1/2^{i+1}$.

$$\begin{aligned}
 & 2^h \times (+i : 0 \leq i \leq h : i/2^i) \\
 \leq & \{\text{arithmetic}\} \\
 & 2^h \times S \\
 = & \{\text{arithmetic}\} \\
 & 2^h \times 2 \cdot (S - S/2) \\
 = & \{\text{from the derivation above}\} \\
 & 2^{h+1} \times (+i : 1 \leq i : 1/2^i) \\
 = & \{\text{sum of the infinite geometric series is 1}\} \\
 & 2^{h+1}
 \end{aligned}$$

Therefore, $(+i : 0 \leq i \leq h : i \cdot 2^{h-i})$ is $\mathcal{O}(2^h)$, that is, $\mathcal{O}(N)$.

Combining the $\mathcal{O}(N \log N)$ cost of $extendHeap$ with $\mathcal{O}(N)$ of $heapify$, the cost of heapsort is $\mathcal{O}(N \log N)$.

Applications of the heap data structure The heap data structure is useful in its own right besides its use in heapsort. It allows removing a maximum (or minimum) element and insertion of a new element into the structure in $\mathcal{O}(\log n)$ time using constant amount of additional space. That is why it is used extensively in simulation programs to store event-lists.

5.6 Knuth–Morris–Pratt String-matching Algorithm

5.6.1 The String-matching Problem and Outline of the Algorithm

We study an interesting algorithm for string matching, to locate one (or all) occurrence of a given *pattern* string within a given *text* string of symbols. The text may be a long string, say the collected works of Shakespeare. The pattern may be a phrase like “to be or not to be”, a word like “Ophelia”, or a long paragraph starting with “Friends, Romans, countrymen, lend me your ears”. If the pattern does not occur in the text, the search fails, otherwise the search shows all, or perhaps the first, position in the text where the pattern occurs.

There are many variations of this basic problem. The pattern may be a set of strings, and the matching algorithm has to locate the occurrence of any of these strings in the text. The set of pattern strings may be infinite, given by a “regular expression”. Quite often, the goal is not to find an exact match but a close enough match, as in searching DNA sequences. The string-matching problem is quite different from dictionary search, where the problem is to determine if a given word belongs to a set of words. Usually, the set of words in the dictionary is fixed. A hashing algorithm suffices in most cases for such problems.

I consider the exact string-matching problem: given a text string T and a pattern string P over some alphabet, find the first position in T where P occurs or indicate that P does not occur in T . For example, given the text and pattern as shown in Table 5.3, the algorithm outputs position 9, the position where the pattern starts in the text. This pattern is also at position 12, and the two matches overlap in the text.

Table 5.3 The string-matching problem

<i>position</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>text</i>	a	a	b	b	a	a	c	a	b	a	a	b	a	a	b	a
<i>pattern</i>	a	a	b	a												

A naive algorithm, described next, may take up to $\mathcal{O}(m \cdot n)$ time to solve the problem where m and n are the lengths of T and P , respectively. The [Knuth–Morris–Pratt \(KMP\)](#) [1977] algorithm is a refinement of the naive algorithm that solves the problem in $\mathcal{O}(m)$ time using extra storage proportional to n . I explain the KMP algorithm as an example of stepwise refinement, first developing a small theory and then deriving a program in steps using the theory. The original KMP algorithm was inspired by a linear-time simulation algorithm of 2-way deterministic push-down automata in [Cook](#) [1970]. The treatment in this section is quite elementary.

There are now much better algorithms than KMP, many based on an algorithm by [Boyer and Moore \[1977\]](#).

5.6.1.1 A Naive Algorithm

At any point during a search, the *left end* refers to the position in text T where the string match starts and *right end* the position up to which the match has succeeded. A naive algorithm starts with both left and right ends at position 0, indicating that no portion of the pattern has been matched against any of the text. It continues matching successive symbols of the pattern against the text, and shifting the right end, until there is a mismatch. Then it attempts to match P anew starting at the next position past the left end of T , that is, shifting the left end by one position to the right and setting the right end to the same position as the left end. It stops only after finding a full match of P or if it runs out of symbols in T , denoting a failure.

To see the worst possible performance of the naive algorithm, consider $T = a^m$, an m -fold repetition of symbol a , and $P = a^n b$, n occurrences of a followed by b , where $n < m$. There will never be a full match. The naive algorithm will start the match at every position in the text (as long as there are still $n + 1$ symbols remaining to match in the text), match the prefix a^n of the pattern using up n computation steps, and find a mismatch at b . The running time is $\mathcal{O}(m \cdot n)$.

The naive algorithm is throwing away a lot of useful information it has gained in a failed match. In the given example, if the left end is i , it will go up to $i + n - 1$ as its right end but fail at the next position in matching b against a . At this point, we know that the prefix of the pattern a^{n-1} matches with left end at $i + 1$ and the right end at $i + n - 1$; so shifting the left end by one position to the right but leaving the right end at its current position is correct. There is no need to match these symbols in the text again. Similarly, if prefix abb of a pattern has already been matched in T and there is a mismatch at the next symbol, we can conclude that there is no full match starting at any of the b 's because the pattern starts with an a ; so, the matching should start anew past this portion of the text.

The KMP algorithm uses the knowledge gained from a failed match, as we explain informally below and formally in the rest of this section.

5.6.1.2 KMP Algorithm

The KMP algorithm runs like the naive algorithm until there is a mismatch. Figure 5.9 shows a schematic of the point when there is a mismatch. Here the text consists of three segments, X , Y and Z , of which the middle segment Y has matched a prefix V of the pattern, so the left end is at the start of Y and the right end at its end. Since we are searching for the first (leftmost) match, we have already scanned

over X , and no position within X is the possible start of a full match. The goal now is to match the pattern beyond V with the text beyond Y .

Suppose there is a mismatch because the next symbol following Y in the text and V in the pattern do not match. Unlike the naive algorithm, KMP precomputes with the pattern so that in case of a mismatch it can *claim* that a certain prefix U of the pattern matches a suffix of Y , as shown. That is, it shifts the left end by some amount to the right but does not alter the right end. Matching of symbols continues with the symbol following the right ends of Y and U . Thus, no symbol within Y , the matched portion of the text, is examined again. Observe that U may be the empty string.

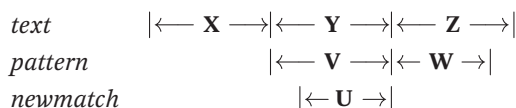


Figure 5.9 Matching in the KMP algorithm.

Determination of U is the crux of the KMP algorithm. As shown in Figure 5.9, U has to satisfy the following properties:

1. U is a proper prefix of V : U is a prefix of the pattern, V is a prefix of the pattern, and V is longer than U .
2. U is a proper suffix of V : U is a suffix of Y and $Y = V$.
3. U is the longest string that satisfies (1) and (2): Proof is by contradiction. Suppose U satisfies (1) and (2) but it is not the longest. Then we may miss a possible match or record a match that is not the leftmost one in the text. I show the first possibility in the following example. Let the text be *aaaab* and the pattern *aaab*, so the pattern matches the text in its right end. Suppose we have matched $V = aaa$ from the left end of the text. The next symbol following V in the pattern, b , does not match the next symbol of the text, a . Now, we have several candidates for U that satisfy (1) and (2): the empty string, a , and aa ; so aa is the required longest string. If, instead, U is taken to be a , the remaining portion of the text to be matched (beyond V) is ab and of the pattern (beyond U) is aab , which would result in a mismatch.

The longest string that is both a prefix and a suffix of V is called the *core* of V in the following development.¹ First, I prove certain properties of core that will enable us to compute the cores of all prefixes of the pattern in linear time in the length

1. The term *lps*, for *longest prefix suffix*, has been used in the literature for *core*.

of the pattern. The computed cores are then used in matching the pattern against the text, which is done in linear time in the length of the text.

Example 5.1 Consider the matching problem given in Table 5.3 (page 222); the pattern and the text are reproduced in Table 5.4. Successive matches during the KMP algorithm are shown in this table. The first attempt starting at position 0, shown in *match*₁, matches up to position 2, and there is a mismatch, shown using the symbol “-”, at position 3. The prefix of the pattern that is matched, “aab”, has an empty core. So, the matching restarts at position 3 from the first symbol of the pattern, fails, and restarts at position 4, as shown in *match*₂. It fails again at position 6. The prefix that has matched, “aa”, has the core “a”, so the pattern is shifted one place to the right and matching resumes at position 6 in the text, and fails as shown in *match*₃. The core of “a”, the prefix that has matched, is empty, so the match starts afresh at position 6, fails and succeeds for one symbol at position 7 and fails at position 8, as shown in *match*₄. Matching starts afresh at 8, fails, and restarts at 9 where it succeeds; see *match*₅.

Matching never restarts at an earlier position in the text.

Table 5.4 Successive matches in the KMP algorithm

<i>position</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>text</i>	a	a	b	b	a	a	c	a	b	a	a	b	a	a	b	a
<i>pattern</i>	a	a	b	a												
<i>match</i> ₁	a	a	b	-												
<i>match</i> ₂					a	a	-									
<i>match</i> ₃						a	-									
<i>match</i> ₄								a	-							
<i>match</i> ₅										a	a	b	a			

5.6.2 Underlying Theory of the KMP Algorithm

Notation I use uppercase letters to denote strings and lowercase ones for symbols of the alphabet over which text T and pattern P are strings. The empty string is written as ε . The length of string U is written as $|U|$.

5.6.2.1 Bifix and Core

Definition String U is a *bifix* of string V , written as $U \preceq V$, if U is both a prefix and a suffix of V . Write $U \prec V$ to mean that U is a proper bifix of V , that is, $U \preceq V$ and $U \neq V$. ■

Observe that $\preceq$ is a total order over the prefixes of a string because both prefix and suffix are total orders. From the reflexivity of total order, every string is its own

bifix, and, ε , the empty string is a bifix of every string. So, every non-empty string has a proper bifix, therefore, a unique longest proper bifix.

Example 5.2 String *abbabb* has one non-empty proper bifix: *abb*. String *aabaaba* has the non-empty proper bifixes *a* and *aaba*. Extending a string does not necessarily extend its proper bifixes; string *abbabbc*, an extension of *abbabb*, has ε as its only bifix. ■

Definition The longest proper bifix of a non-empty string V is called its *core* and written as $c(V)$. Formally, define $c(V)$ by: $U \preceq c(V) \equiv U \prec V$. ■

The traditional way to define $c(V)$ is as follows: (1) $c(V) \prec V$, and (2) for any W where $W \prec V$, $W \preceq c(V)$. Our formal definition of core looks unusual, yet it is more convenient for formal manipulations. For example, by replacing U with $c(V)$ in the formal definition, we have $c(V) \preceq c(V) \equiv c(V) \prec V$, which implies $c(V) \prec V$, that is, that $c(V)$ is a proper bifix of V .

Further, we can show that $c(V)$ is unique for any non-empty V . To see this, let R and S be cores of V . Using R and S for $c(V)$ in the formal definition, we get $U \preceq R \equiv U \prec V$ and $U \preceq S \equiv U \prec V$. That is, $U \preceq R \equiv U \preceq S$, for all U . Setting U to R , we get $R \preceq R \equiv R \preceq S$, that is, $R \preceq S$. Similarly, we can deduce $S \preceq R$. So, conclude $R = S$ because $\preceq$ is a total order.

Given uniqueness of the core, treat c as a function over non-empty strings. Write $c^i(V)$ for the i -fold application of c to V , that is, $c^0(V) = V$ and $c^{i+1}(V) = c(c^i(V))$. Since $|c(V)| < |V|$, $c^i(V)$ is defined only for some i , not necessarily all i , where $0 \leq i \leq |V|$. From $c(V) \prec V$, $c^{i+1}(V) \prec c^i(V)$ for all i whenever $c^{i+1}(V)$ is defined.

5.6.2.2 A Characterization of Bifixes

Theorem 5.1 String U is a bifix of V iff for some i , $i \geq 0$, $U = c^i(V)$.

Proof. Proof is by mutual implication.

• Show that $c^i(V) \preceq V$ for all $i \geq 0$:

Proof is by induction on i , using the fact that $c(W) \prec W$ for any non-empty string W .

Base case, $i = 0$: $c^0(V) = V \preceq V$.

Inductive case: Suppose $U = c^{i+1}(V)$.

$$\begin{aligned}
 & c^{i+1}(V) \\
 \prec & \{c^{i+1}(V) = c(c^i(V)). \text{ Let } W = c^i(V), \text{ so } c^{i+1}(V) = c(W) \prec W = c^i(V)\} \\
 & c^i(V) \\
 \preceq & \{\text{inductively } c^i(V) \preceq V\} \\
 & V
 \end{aligned}$$

- Given $U \preceq V$, show that there is some $i, i \geq 0$, such that $U = c^i(V)$:

Proof is by induction on the length of V .

Base case, $|V| = 0$:

$$\begin{aligned}
 & U \preceq V \\
 \equiv & \{ |V| = 0, \text{ i.e., } V = \varepsilon \} \\
 & U = \varepsilon \wedge V = \varepsilon \\
 \equiv & \{ c^0(\varepsilon) = \varepsilon, \text{ and } c^i(\varepsilon) \text{ is defined for } i = 0 \text{ only} \} \\
 & (\exists i : i \geq 0 : U = c^i(V))
 \end{aligned}$$

Inductive case, $|V| > 0$:

$$\begin{aligned}
 & U \preceq V \\
 \equiv & \{ \text{definition of } \preceq \} \\
 & U = V \vee U \prec V \\
 \equiv & \{ \text{definition of core} \} \\
 & U = V \vee U \preceq c(V) \\
 \equiv & \{ |c(V)| < |V|; \text{ apply induction hypothesis on second term} \} \\
 & U = V \vee (\exists i : i \geq 0 : U = c^i(c(V))) \\
 \equiv & \{ \text{rewrite } U = V \text{ as } U = c^0(V) \} \\
 & (\exists i : i \geq 0 : U = c^i(V))
 \end{aligned}$$

Corollary 5.1 $U \prec V \equiv (\exists i : i > 0 : U = c^i(V)).$

Proof. Directly from the theorem. ■

5.6.3 Core Computation

The informal description of the KMP algorithm shows that the core of any prefix of the pattern may be needed during the matching. The algorithm computes the cores of *all* prefixes of the pattern before starting a match, and this computation takes linear time in the length of the pattern.

5.6.3.1 Underlying Theorem for Core Computation

Notation For string U that is a proper prefix of a known string V , write $next(U)$ for the symbol following U in V . Write $V.s$ for V followed by symbol s .

Theorem 5.2 String $U.t$ is the core of $V.s$ iff for some $i, i > 0$,

- (1) $U = c^i(V)$ and $t = s = next(U)$, and
- (2) i is the smallest value for which (1) holds.

If there is no i for which (1) holds, $c(V.s) = \varepsilon$.

Proof. First, consider any proper bifix $U.t$ of $V.s$.

$$\begin{aligned}
& U.t \prec V.s \\
\equiv & \{\text{definition of } \prec\} \\
& U.t \text{ is a proper prefix of } V.s, U.t \text{ is a proper suffix of } V.s \\
\Rightarrow & \{U.t \text{ is a proper prefix of } V.s \text{ implies } U \text{ is a proper prefix of } V; \\
& U.t \text{ is a proper suffix of } V.s \text{ iff } U \text{ is a proper suffix of } V \text{ and } t = s\} \\
& U \text{ is a proper prefix of } V, t = s, U \text{ is a proper suffix of } V \\
\equiv & \{\text{definition of } \prec\} \\
& U \prec V, t = s \\
\equiv & \{\text{Corollary 5.1: } U \prec V \equiv (\exists i : i > 0 : U = c^i(V))\} \\
& (\exists i : i > 0 : U = c^i(V)), t = s \\
\equiv & \{\text{definition of } next\} \\
& (\exists i : i > 0 : U = c^i(V)), next(U) = s
\end{aligned}$$

The core is the longest non-empty proper bifix, so i is as small as possible in the derivation above. And, if there is no i for which (1) in the theorem holds, $c(V.s) = \varepsilon$. ■

5.6.3.2 Abstract Program for Core Computation

The main loop of the program in Figure 5.10 (page 229) computes the cores of all prefixes of pattern P . In an iteration, it computes the core of a prefix $V.s$ of P . It assumes that the cores of all prefixes of V have already been computed. During the computation for $V.s$, the values of $c^i(V)$, for various i , are needed. The program simply refers to them; later refinements show how they are computed.

From Theorem 5.2, $c(V.s)$ is $c^i(V).s$ for the smallest i such that $next(c^i(V)) = s$, or ε if there is no such i . The inner loop of the program in Figure 5.10 compares $next(c^i(V))$ and s for successively larger i , starting with $i = 1$. Eventually, either (1) $next(c^i(V)) = s$ for some i , so $core[V.s]$ is $c^i(V).s$, or (2) there is no such i , so $core[V.s] = \varepsilon$.

The outer loop uses invariant I_0 and the inner loop I_1 :

$I_0 :: V$ is a non-empty prefix of P , $(\forall U : U \text{ a prefix of } V : core[U] = c(U))$.

$I_1 :: I_0, next(V) = s, i > 0, (\forall j : 1 \leq j < i : next(c^j(V)) \neq s)$.

The proofs of the verification conditions are entirely straightforward using Theorem 5.2. At the end of the computation, the cores of all prefixes of P are available in array *core*.

I do not prove termination of this program. I prove a linear running time for the program in Section 5.6.3.4 (page 230), thus establishing termination.

Core computation1

```

V, core[V] := next( $\epsilon$ ),  $\epsilon$ ;
{ I0 }
while V ≠ P do
  { I0, V ≠ P }
  s, i := next(V), 1;
  { I1 }
  while next(ci(V)) ≠ s ∧ ci(V) ≠  $\epsilon$  do
    { I1, next(ci(V)) ≠ s ∧ ci(V) ≠  $\epsilon$  }
    i := i + 1
    { I1 }
  enddo ;
  { I1, next(ci(V)) = s ∨ ci(V) =  $\epsilon$  }
  if next(ci(V)) = s
    then { I1, next(ci(V)) = s } V, core[V.s] := V.s, ci(V).s { I0 }
    else { I1, next(ci(V)) ≠ s ∧ ci(V) =  $\epsilon$  } V := V.s; core[V.s] :=  $\epsilon$  { I0 }
  endif
  { I0 }
enddo
{ I0, V = P }

```

Figure 5.10 Abstract program for core computation.**5.6.3.3 Representation of Prefixes**

The next refinement, program **Core computation2** of Figure 5.11, represents each prefix of *P* and their cores in a more efficient fashion, simply by their lengths. Introduce index variable *v* corresponding to prefix *V* so that *V* = *P*[0..*v*], and represent *V.s* by *v* + 1 and *next*(*V*) by *P*[*v*]. Then ϵ is represented by 0 and the entire pattern *P* of length *n* by *n*. Now *core*[1..*n*] is an array such that *core*[*v*] is the length of *c*(*V*).

The invariants are refined as follows; I use variable *V* as before. The core function, *core*, is applied to a prefix length instead of a prefix, and I introduce *civ* for the length of *c*^{*i*}(*V*):

$$I'_0 :: 1 \leq v \leq n, (\forall u : u \leq v : U = P[0..u] \Rightarrow \text{core}[u] = |c(U)|).$$

$$I'_1 :: I, V = P[0..v], P[v] = s, \text{civ} = |c^i(V)|, (\forall j : 1 \leq j < i : \text{next}(c^j(V)) \neq s).$$

Core computation2

```

 $v, core[v] := 1, 0;$ 
 $\{ I'_0 \}$ 
while  $v \neq n$  do
   $\{ I'_0, v \neq n \}$ 
   $s, civ := P[v], core[v];$ 
   $\{ I'_1 \}$ 
  while  $P[civ] \neq s \wedge civ \neq 0$  do
     $\{ I'_1, next(c^i(V)) \neq s \wedge c^i(V) \neq \varepsilon \}$ 
     $civ := core[civ]$ 
     $\{ I'_1 \}$ 
  enddo ;
   $\{ I'_1, P[civ] = s \vee civ = 0 \}$ 
  if  $P[civ] = s$ 
    then  $\{ I'_1, P[civ] = s \} \ v := v + 1 ; core[v] := civ + 1 \ \{ I'_0 \}$ 
    else  $\{ I'_1, P[civ] \neq s \wedge civ = 0 \} \ v := v + 1 ; core[v] := 0 \ \{ I'_0 \}$ 
  endif
   $\{ I'_0 \}$ 
enddo
 $\{ I'_0, v = n \}$ 

```

Figure 5.11 Refinement of the program in Figure 5.10.**5.6.3.4 Running Time**

It is not immediately clear that the core computation algorithm has a linear running time. Variable v only increases, but civ both decreases and increases. Observe that whenever civ increases it increases by 1, and it is accompanied by an increase in v by 1. So, the metric $civ - 2 \cdot v$ always decreases, and it can only decrease $\mathcal{O}(n)$ times, as I show below.

To make this argument precise, transform program **core-computation2** in some minor ways: (1) initialize civ before the main loop, so $civ - 2 \cdot v$ is always defined, and (2) remove the assignment $s, civ := P[v], core[v]$ by replacing s by $P[v]$ in the program and executing $civ := core[v]$ at the end of each iteration of the outer loop. The transformed program, **Core computation**, is shown without any annotation in

Core computation

```

v, core[v], civ := 1, 0, 0;
while v ≠ n do
  while P[civ] ≠ P[v] ∧ civ ≠ 0 do
    civ := core[civ]
  enddo ;
  if P[civ] = P[v]
  then
    v := v + 1; core[v] := civ + 1; civ := core[v]
  else
    v := v + 1; core[v] := 0; civ := core[v]
  endif
enddo

```

Figure 5.12 Core computation for all prefixes of *P*.

Figure 5.12 (page 231). It is easy to see that:

1. The initial value of $civ - 2 \cdot v$ is -2 .
2. The final value of $civ - 2 \cdot v$ is $civ - 2 \cdot n \geq \{civ \geq 0\} - 2 \cdot n$.
3. Each sequence of assignment commands strictly decreases $civ - 2 \cdot v$. Prove this result using $civ \geq 0$ and $core[civ] < civ$.

From the above facts, the total number of steps executed by the program is at most $-2 - (-2 \cdot n)$, that is, $\mathcal{O}(n)$.

5.6.4 KMP Program

Refer to the informal description of the KMP algorithm in Section 5.6.1.2 (page 223). In the abstract program, we use XY (concatenation of X and Y) for the prefix of text T whose suffix V matches a prefix of pattern P (see Figure 5.9, page 224). The invariant of the program is:

$J :: XY$ is a prefix of T , V is a prefix of P , V matches a suffix of XY ,
there is no match starting at any prior position in XY .

The invariant is easy to prove using the properties of core of V , written as $c(V)$ in this program. The loop terminates if it runs out of the text ($XY = T$), so there is no

match, or the pattern matches ($V = P$). From invariant J , P is a suffix of XY , so the starting position of the match is $|XY| - |V|$.

Abstract KMP program Partial correctness of **KMP1** of Figure 5.13 follows from the annotations. I show a linear running time for the concrete program **KMP2**.

KMP1

```

 $XY, V := \epsilon, \epsilon;$ 
{  $J$  }
while  $XY \neq T \wedge V \neq P$  do
  if  $next(XY) = next(V)$ 
  then { next symbols in  $T$  and  $P$  match; extend  $XY$  and  $V$  }
     $XY, V := XY.next(XY), V.next(V)$ 
  else
    if  $V = \epsilon$ 
    then { empty pattern has matched }  $XY := XY.next(XY)$ 
    else {  $V$  is non-empty, so has a core }  $V := c(V)$ 
    endif
  endif
enddo ;
{  $J, XY = T \vee V = P$  }
if  $XY = T$ 
then “report there is no match”
else {  $V = P$  } “report a match at position  $|XY| - |V|$  of  $T$ ”
endif

```

Figure 5.13 Abstract KMP program.

Concrete KMP program Figure 5.14 shows a refinement of program **KMP1** of Figure 5.13. As we did for the core computation, represent each string by its length (and its name by the corresponding lowercase letters), and use array *core* that is computed by the program **Core computation** in Figure 5.12 (page 231).

For deriving the running time, note that each iteration of the loop either (1) increases both xy and v , (2) increases xy while keeping v the same, or (3) decreases v while keeping xy the same. Therefore, $2 \cdot xy - v$ increases in each iteration. This quantity is at most $2 \cdot m$, so the program has a running time of $\mathcal{O}(m)$.

KMP2

```

 $xy, v := 0, 0;$ 
 $\{ J \}$ 
while  $xy \neq m \wedge v \neq n$  do
  if  $T[xy] = P[v]$ 
    then  $\{ \text{next symbols in } T \text{ and } P \text{ match; extend } xy \text{ and } v \}$ 
        $xy, v := xy + 1, v + 1$ 
    else
      if  $v = 0$ 
        then  $\{ \text{empty pattern has matched} \}$   $xy := xy + 1$ 
        else  $\{ v \text{ represents a non-empty prefix, which has a core} \}$   $v := \text{core}[v]$ 
      endif
    endif
  enddo ;
 $\{ J, xy = m \vee v = n \}$ 
if  $xy = m$ 
  then “report there is no match”
else  $\{ v = n \}$  “report a match at position  $xy - v$  of  $T$ ”
endif

```

Figure 5.14 Concrete KMP program.

5.7 A Sieve Algorithm for Prime Numbers

This section illustrates refinement of a program in stages, applied to the problem of finding all prime numbers up to a given limit. Except for its pedagogical value, the problem and its solution have no utility in practice since tables of primes for all practical needs have already been computed; I just looked up all primes up to one trillion (10^{12}) online. The solution given in this section, however, illustrates the program analysis techniques we have seen in the previous chapters. The solution starts with the classic, and classical, sieve algorithm of Eratosthenes of ancient Greece (perhaps 3rd century BCE). The refinements are guided by mathematical properties of prime numbers. Invariant assertions and program correctness play essential roles throughout the development. The development here follows [Misra \[1981\]](#).

A *prime number* is an integer, at least 2, that is divisible only by 1 and itself. Henceforth, a non-prime integer, at least 2, is called a *composite*. A *multiple* of

integer p , $2 \leq p$, is any $p \cdot t$ where t is integer and $2 \leq t$; so, a multiple is always composite.

The method of Eratosthenes for finding all primes up to n starts with set P of integers from 2 to n . Take the smallest number p in P , initially p is 2. Remove all multiples of p from P . Now set p to the next larger number in P , which is now 3, and remove all its multiples from P . Repeat this step until P has no next larger number. The numbers remaining in P at termination are all the prime numbers less than or equal to n .

An intuitive justification of the algorithm uses the fact that any number that is removed is a multiple of some number, hence a composite. What is not clear is that *all* composites are actually removed, a fact that is easier to prove formally. The removal of composites resembles a sieve that casts out the multiples of a prime in each iteration.

I describe the method of Eratosthenes more formally in the next section and prove its correctness. Then I develop a precise characterization of the composites and, based on this characterization, refine the original sieve algorithm.

5.7.1 Sieve of Eratosthenes

Terminology and convention All variables in this section are integers greater than or equal to 2; $prime(x)$ denotes x is prime and $\neg prime(x)$ that x is composite. Variable P is a set that includes all primes up to n , initially containing all numbers from 2 to n and finally containing exactly the prime numbers from this set. Write $succ(x)$, for x in P , for the next larger number than x in P ; it is $n + 1$ if x is the largest number in P . Similarly, $pred(x)$ is the next number smaller than x in P , and it is 1 if x is the smallest number in P . Extend the definition of $succ$ so that $succ(1) = 2$, in order to simplify certain proofs. ■

The goal is to compute P where $P = \{x \mid 2 \leq x \leq n, prime(x)\}$, for any given n , $n \geq 2$. The program in Figure 5.15 uses the Eratosthenes sieve transcribed in a modern notation. It includes an informal command whose meaning is specified by its pre- and postcondition, where the auxiliary variable P_0 retains the value of P before the command execution.

5.7.2 Correctness

Let $sf(x)$ be the *smallest factor* of x , that is, the smallest number other than 1 that divides x with no remainder. For x and y , $x \geq 2$, $y \geq 2$, define sf by:

- (1) $prime(x) \Rightarrow sf(x) = x$,
- (2) $sf(x \cdot y) = \min(sf(x), sf(y))$.

Eratosthenes-Sieve

```

P := {x | 2 ≤ x ≤ n};
p := 2;
while p < n do
  {P = P0}
  remove all multiples of p from P;
  {P = P0 − {p · t | t ≥ 2}}
  p := succ(p)
enddo
{ P = {x | 2 ≤ x ≤ n, prime(x)} }

```

Figure 5.15 Sieve of Eratosthenes in modern notation.

Note the following properties of *sf* whose proofs use elementary number theory: For any *x*, (1) *prime*(*x*) ≡ (*sf*(*x*) = *x*), (2) *x* ≥ *sf*(*x*), (3) for composite *x*, *x* ≥ *sf*²(*x*), and (4) for all *x*, *sf*(*sf*(*x*)) = *sf*(*x*).

5.7.2.1 Invariant

After removal of the multiples of *p* in **Eratosthenes-Sieve** of Figure 5.15, no element *x* of *P*, other than *p*, has *sf*(*x*) = *p*. The following invariant codifies this property:

$$I :: (p \leq n \equiv p \in P), P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee \text{sf}(x) \geq p\}$$

Note that *I* also specifies a property of the numbers outside *P*:

$$(x \notin P \wedge 2 \leq x \leq n) \Rightarrow (\neg \text{prime}(x) \wedge \text{sf}(x) < p).$$

It follows from *I* that $p \leq n \Rightarrow p \in P$. And, we derive that *p* is a prime:

$$\begin{aligned}
 & p \in P \\
 \Rightarrow & \{\text{from the definition of } P \text{ in } I\} \\
 & \text{prime}(p) \vee \text{sf}(p) \geq p \\
 \equiv & \{\text{property of sf: } p \geq \text{sf}(p)\} \\
 & \text{prime}(p) \vee \text{sf}(p) = p \\
 \Rightarrow & \{\text{property of sf: } \text{prime}(p) \equiv (\text{sf}(p) = p)\} \\
 & \text{prime}(p) \vee \text{prime}(p) \\
 \Rightarrow & \{\text{predicate calculus}\} \\
 & \text{prime}(p)
 \end{aligned}$$

Note Invariant I does not claim that $p \leq n \wedge p \in P$. This is because the execution of $p := \text{succ}(p)$ in the last iteration may violate both conjuncts. ■

5.7.2.2 Verification Conditions

Figure 5.16 shows an annotated version of the program in Figure 5.15. I do not formally specify the meaning of succ . I only show that a precondition of $p := \text{succ}(p)$ is $p \in P$. Further, I will use the fact that $p < \text{succ}(p)$ and $(\forall p' : p < p' < \text{succ}(p) : p' \notin P)$, that is, values strictly between p and $\text{succ}(p)$ are outside P .

Annotated-Eratosthenes-Sieve

```

{ true }
P := {x | 2 ≤ x ≤ n};
p := 2;
{I}
while p ≤ n do
  {I, p ≤ n, p ∈ P, P = P0}
  remove all multiples of p from P;
  { p ≤ n, p ∈ P, P0 = {x | 2 ≤ x ≤ n, prime(x) ∨ sf(x) ≥ p} ,
    P = P0 - {p · t | t ≥ 2} }
  { p ≤ n, p ∈ P, P = {x | 2 ≤ x ≤ n, prime(x) ∨ sf(x) > p} }
  p := succ(p)
  { I }
enddo
{ I, p > n }
{ P = {x | 2 ≤ x ≤ n, prime(x)} }
```

Figure 5.16 Annotation of the program in Figure 5.15.

It is easy to generate and prove most of the verification conditions. I treat the two cases that need special attention.

Case 1)

```

{ p ≤ n, p ∈ P, P = {x | 2 ≤ x ≤ n, prime(x) ∨ sf(x) > p} }
p := succ(p)
{I}
```

The corresponding verification condition, expanding I , is:

$$\begin{aligned}
I' &:: p \leq n, p \in P, P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee \text{sf}(x) > p\} \\
\Rightarrow \\
p \leq n &\equiv p \in P, P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee \text{sf}(x) \geq \text{succ}(p)\}
\end{aligned}$$

It is easy to prove $p \leq n \equiv p \in P$ in the consequent; I only show $P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee \text{sf}(x) \geq \text{succ}(p)\}$. Consider any value x in P for which $\text{sf}(x) > p$. I show that $\text{sf}(x) \geq \text{succ}(p)$, that is, $\text{sf}(x)$ does not fall between p and $\text{succ}(p)$.

$$\begin{aligned}
& I', x \in P, \text{sf}(x) > p \\
\Rightarrow & \{\text{definition of succ}\} \\
& I', x \in P, \text{sf}(x) \notin P \vee \text{sf}(x) \geq \text{succ}(p) \\
\Rightarrow & \{\text{using sf}(x) \text{ for } x \text{ in } I', \text{sf}(x) \notin P \Rightarrow \text{sf}(\text{sf}(x)) \leq p \Rightarrow \text{sf}(x) \leq p\} \\
& I', x \in P, \text{sf}(x) \leq p \vee \text{sf}(x) \geq \text{succ}(p) \\
\Rightarrow & \{(I', x \in P, \text{sf}(x) \leq p) \Rightarrow \text{prime}(x)\} \\
& I', \text{prime}(x) \vee \text{sf}(x) \geq \text{succ}(p)
\end{aligned}$$

Case 2) I derive the postcondition from $I \wedge p > n$, that is, I show $(I \wedge p > n) \Rightarrow (P = \{x \mid 2 \leq x \leq n, \text{prime}(x)\})$.

$$\begin{aligned}
& I \wedge p > n \\
\Rightarrow & \{\text{expanding parts of } I\} \\
& P = \{x \mid 2 \leq x \leq n \wedge (\text{prime}(x) \vee \text{sf}(x) \geq p)\}, p > n \\
\Rightarrow & \{\text{no number } x, x \leq n, \text{ has sf}(x) > n; \text{ so } \neg(\text{sf}(x) \geq p)\} \\
& P = \{x \mid 2 \leq x \leq n \wedge (\text{prime}(x) \vee \text{sf}(x) \geq p) \wedge \neg(\text{sf}(x) \geq p)\} \\
\Rightarrow & \{\text{predicate calculus}\} \\
& P = \{x \mid 2 \leq x \leq n, \text{prime}(x)\}
\end{aligned}$$

5.7.2.3 Termination

Variable p strictly increases in each iteration because $\text{succ}(p)$ is greater than p for all values of $p, p \in P$. Therefore, the condition for iteration, $p \leq n$, will become *false* eventually.

5.7.3 Characterization of Composite Numbers

Program **Eratosthenes-Sieve** of Figure 5.15 (page 235) is easy to state and relatively easy to prove (though we can be sure that Eratosthenes never proved its correctness). It is not very efficient though. It attempts to remove some numbers more than once, such as 6 which is removed for $p = 2$ and again for $p = 3$. In fact, every composite is removed as many times as the number of its prime factors.

In this section I develop a theorem that permits us to remove every composite exactly once. The theorem characterizes all multiples of p that are still in P .

Theorem 5.3 Given $I \wedge p \leq n$:

- (1) there is a unique q in P such that $p \cdot \text{pred}(q) \leq n < p \cdot q$, and
- (2) the set of multiples of p in P is $\{p \cdot t \mid p \leq t < q, t \in P\}$.

Proof of part 1) There is a unique q in P such that $p \cdot \text{pred}(q) \leq n < p \cdot q$:
 Note that $\text{pred}(q)$ need not be in P ; in particular if $q = 2$, $\text{pred}(q) = 1$.

- There is a number u in P such that $p \cdot u > n$: I use a deep theorem of number theory, postulated by Bertrand and first proved by Chebyshev (see [LeVeque \[1956\]](#)):
 For any integer u , $u > 1$, there is a prime number v such that $u < v < 2 \cdot u$.

It follows that:

If u and v are adjacent prime numbers where $u < v$, then $v < 2 \cdot u$. (B-C)

Let u be the largest prime less than or equal to n and v the next larger prime; so $v > n$.

$$\begin{aligned}
 & v > n \\
 \Rightarrow & \{v < 2 \cdot u \text{ from (B-C)}\} \\
 & 2 \cdot u > n \\
 \Rightarrow & \{\text{from } I \wedge p \leq n, \text{ as we have shown, } p \in P; \text{ so } p \geq 2\} \\
 & p \cdot u > n
 \end{aligned}$$

- There is a unique q in P such that $p \cdot \text{pred}(q) \leq n < p \cdot q$:
 Let u be such that $p \cdot u > n$. Consider the sequence of values $\langle p \cdot 1, p \cdot 2 \dots p \cdot \text{pred}(x), p \cdot x \dots p \cdot u \rangle$, for all x in P . This sequence is increasing because $\text{pred}(x) < x$. Given that $p \cdot 1 = p \leq n$ and $p \cdot u > n$, there is a unique q in P for which $p \cdot \text{pred}(q) \leq n < p \cdot q$.

Proof of part 2) For any t , $t \geq 2$, $p \cdot t \in P \equiv (p \leq t < q \wedge t \in P)$:

Proof is by mutual implication.

- $p \cdot t \in P \Rightarrow (p \leq t < q \wedge t \in P)$:

$$\begin{aligned}
 & p \cdot t \in P \\
 \Rightarrow & \{\text{from the definition of } P, p \cdot t \leq n. \text{ So, } t < q\} \\
 & p \cdot t \in P, t < q \\
 \Rightarrow & \{\neg \text{prime}(p \cdot t) \text{ and } p \cdot t \in P. \text{ Using } I\} \\
 & \text{sf}(p \cdot t) \geq p, t < q \\
 \Rightarrow & \{\text{from properties of sf, sf}(p \cdot t) = \min(\text{sf}(p), \text{sf}(t))\} \\
 & \text{sf}(t) \geq p, t < q \\
 \Rightarrow & \{\text{from properties of sf, } t \geq \text{sf}(t)\} \\
 & t \geq \text{sf}(t), \text{sf}(t) \geq p, t < q \\
 \Rightarrow & \{\text{rewriting}\} \\
 & \text{sf}(t) \geq p, p \leq t < q \\
 \Rightarrow & \{p \leq t < q \text{ implies } 2 \leq t \leq n. \text{ Using sf}(t) \geq p \text{ with } I, t \in P\} \\
 & p \leq t < q, t \in P
 \end{aligned}$$

$$\begin{aligned}
& \bullet (p \leq t < q \wedge t \in P) \Rightarrow p \cdot t \in P : \\
& \quad p \leq t < q \wedge t \in P \\
& \Rightarrow \{ \text{from } t \in P \text{ and } I, \text{prime}(t) \vee \text{sf}(t) \geq p \} \\
& \quad p \leq t < q \wedge (\text{prime}(t) \vee \text{sf}(t) \geq p) \\
& \Rightarrow \{ \text{prime}(t) \Rightarrow \text{sf}(t) = t; \text{from } t \geq p, \text{sf}(t) \geq p \} \\
& \quad p \leq t < q \wedge \text{sf}(t) \geq p \\
& \Rightarrow \{ \text{sf}(p \cdot t) = \min(\text{sf}(p), \text{sf}(t)). \text{ From } I \wedge p \leq n, p \text{ is prime, so } \text{sf}(p) = p \} \\
& \quad p \leq t < q, \text{sf}(p \cdot t) = p \\
& \Rightarrow \{ p \leq p \cdot t, \text{from } t \geq 2. \text{ And } p \cdot t \leq n, \text{from } t < q \} \\
& \quad p \leq p \cdot t \leq n, \text{sf}(p \cdot t) = p \\
& \Rightarrow \{ \text{from } I \} \\
& \quad p \cdot t \in P
\end{aligned}$$

Observe from part (2) of the theorem that if $p \geq q$, then p has no multiples in P . And, if $p < q$, then $p \leq p < q$, so $p^2 \in P$.

5.7.4 Refinement of the Sieve Program

Theorem 5.3 provides the ingredients for a refinement of program **Eratosthenes-Sieve**, Figure 5.15, which I show in Figure 5.17. I retain the previous invariant I , and, motivated by Theorem 5.3, I introduce invariant J that defines q . I will derive an appropriate iteration condition for the loop as part of the refinement.

$$I :: P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee \text{sf}(x) \geq p\},$$

$$J :: q \in P \wedge p \cdot \text{pred}(q) \leq n < p \cdot q.$$

5.7.4.1 Initialization

Variables P and p are initialized as in program **Eratosthenes-Sieve** of Figure 5.15 (page 235) to establish invariant I . For invariant J , set q to $\lfloor n/2 \rfloor + 1$. Verify that for any n , $n \geq 2$: (i) $\lfloor n/2 \rfloor + 1$ is in P , (ii) $\text{pred}(\lfloor n/2 \rfloor + 1) = \lfloor n/2 \rfloor$, and (iii) $2 \cdot \lfloor n/2 \rfloor \leq n < 2 \cdot (\lfloor n/2 \rfloor + 1)$.

$$\begin{aligned}
& \{ \text{true} \} \\
& P := \{x \mid 2 \leq x \leq n\}; \\
& p := 2; \\
& q := \lfloor n/2 \rfloor + 1 \\
& \{I \wedge J\}
\end{aligned}$$

5.7.4.2 Loop Iteration Condition

Program **Eratosthenes-Sieve** of Figure 5.15 uses $p \leq n$ as the condition for continuation of loop iterations. I prove that $p < q$ can be used instead. Informally, if $p \geq q$, then, using Theorem 5.3, there is no multiple of p in P . In the next iteration, p is increased by the assignment $p := \text{succ}(p)$. Therefore, q remains the same or decreases to preserve invariant J , so $p \geq q$ continues to hold, and no multiple is

Eratosthenes-Sieve Refined

```

{ true }
initialize  $P$ ,  $p$  and  $q$ ;
 $\{I \wedge J\}$ 
while  $b$  do
   $\{p \in P, I \wedge J \wedge b, P = P_0\}$ 
  remove all multiples of  $p$  from  $P$ ;
   $\{p \in P, P_0 = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee sf(x) \geq p\},$ 
     $P = P_0 - \{p \cdot t \mid t \geq 2\}\}$ 
   $\{p \in P, P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee sf(x) > p\}\}$ 
   $p := \text{succ}(p)$ ;
   $\{I\}$ 
  reestablish  $J$ ;
   $\{I \wedge J\}$ 
enddo ;
 $\{I \wedge J \wedge \neg b\}$ 
 $\{P = \{x \mid 2 \leq x \leq n, \text{prime}(x)\}\}$ 

```

Figure 5.17 Refinement of the program in Figure 5.15.

ever removed. It is easier to use a formal argument to arrive at the same result, as shown in Theorem 5.4.

Theorem 5.4 Given $I \wedge J$, P includes a composite iff $p < q$.

Proof. Theorem 5.3 shows that if $p < q$, then P includes a composite because there is some t satisfying $p \leq t < q$. In particular, with $t = p$, p^2 is included in P . Below, I prove the converse: given $I \wedge J$ and that there is a composite x in P , $p < q$.

$$\begin{aligned}
& x \in P, \neg \text{prime}(x) \\
\Rightarrow & \{\text{from } I\} \\
& x \in P, \neg \text{prime}(x), sf(x) \geq p \\
\Rightarrow & \{\text{property of } sf \text{ for composites: } x \geq sf^2(x)\} \\
& x \in P, x \geq p^2 \\
\Rightarrow & \{x \in P \Rightarrow x \leq n\} \\
& p^2 \leq n \\
\Rightarrow & \{p \cdot q > n, \text{ from } J, \text{ and } p^2 \leq n. \text{ So, } p^2 < p \cdot q\} \\
& p < q
\end{aligned}$$

■

Corollary 5.2 $I \wedge J \wedge p < q$ implies $p \in P \wedge p \leq \sqrt{n}$.

Proof. From the theorem, $p < q$ implies there is a composite in P , and a proof step shows that $p^2 \leq n$, so $p \leq \sqrt{n}$. From $I, p \leq n \Rightarrow p \in P$. ■

To establish the postcondition of the program, I show

Corollary 5.3 $I \wedge J \wedge p \geq q$ implies $P = \{x \mid 2 \leq x \leq n, \text{prime}(x)\}$.

Proof. From the theorem, $J \wedge p \geq q$ implies that there is no composite in P . Then from $I, P = \{x \mid 2 \leq x \leq n, \text{prime}(x)\}$. ■

5.7.4.3 Removing Multiples of p , Updating p

According to Theorem 5.3 removal of the multiples of p requires enumerating all values t between p and q , including p but excluding q , and removing $p \cdot t$ iff $t \in P$. Given $I \wedge J$, both p and q are in P , so we can repeatedly use *succ* starting at p or *pred* starting at q to enumerate t in increasing or decreasing order; this strategy avoids testing t for membership in P . There is a problem with the strategy of enumerating t in increasing order; we may miss removing certain composites. For example, with $p = 3$, first $p \cdot p = 9$ is removed, so $p \cdot 9 = 27$ is never removed. This problem does not arise if t is enumerated in decreasing order, starting at q and repeatedly applying *pred*.

Introduce variable r that is initially set to q . Value of r is decreased in each iteration by applying *pred* and removing $p \cdot r$ until r becomes p ; see Figure 5.18. The required invariant for the correctness of this loop is:

$$K :: P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee sf(x) \geq p\} - \{p \cdot t \mid r \leq t < q, t \in P\}, p \leq r$$

Removing multiples of p , updating p

```

 $r := q;$ 
 $\{ K \wedge p \leq r \}$ 
while  $p \neq r$  do
   $\{ K \wedge p < r \}$ 
   $r := \text{pred}(r);$ 
   $P := P - \{p \cdot r\}$ 
   $\{ K \wedge p \leq r \}$ 
enddo ;
 $\{ K \wedge p = r \}$ 
 $\{ P := \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee sf(x) \geq p\} - \{p \cdot t \mid p \leq t < q, t \in P\} \}$ 
 $\{ P = \{x \mid 2 \leq x \leq n, \text{prime}(x) \vee sf(x) > p\}, \text{ from Theorem 5.3} \}$ 
 $p := \text{succ}(p)$ 
 $\{ I \}$ 

```

Figure 5.18

The loop terminates because r decreases in each iteration and it cannot decrease below p .

5.7.4.4 Reestablishing Invariant J

Removing multiples of p and executing $p := \text{succ}(p)$ will very likely destroy the invariant $J :: q \in P \wedge p \cdot \text{pred}(q) \leq n < p \cdot q$, possibly because q may be removed, so $q \in P$ may not hold, or the execution of $p := \text{succ}(p)$ may invalidate $p \cdot \text{pred}(q) \leq n$. So, invariant J has to be reestablished. I consider two different techniques, recomputing q directly or recomputing $\text{pred}(q)$ from which q is computed by applying succ .

Recomputing q First, observe that $p \cdot \lfloor n/p \rfloor \leq n < p \cdot (\lfloor n/p \rfloor + 1)$. This inequality asserts that q exceeds $\lfloor n/p \rfloor$. So, test successively larger values of q beyond $\lfloor n/p \rfloor$ to find the first value in P for which $n < p \cdot q$ holds.

```

 $q := \lfloor n/p \rfloor + 1;$ 
 $\{p \cdot \lfloor n/p \rfloor \leq n < p \cdot q\}$ 
 $\{(\forall q' : q' < q, q' \in P : p \cdot q' \leq n), n < p \cdot q\}$ 
while  $q \notin P$  do
   $\{(\forall q' : q' < q, q' \in P : p \cdot q' \leq n), n < p \cdot q, q \notin P\}$ 
   $q := q + 1$ 
   $\{(\forall q' : q' < q, q' \in P : p \cdot q' \leq n), n < p \cdot q\}$ 
enddo
 $\{(\forall q' : q' < q, q' \in P : p \cdot q' \leq n), n < p \cdot q, q \in P\}$ 
 $\{J\}$ 

```

I show an upper bound on the execution time of this program fragment over the entire computation. Unfortunately, it is not $\mathcal{O}(n)$ but $\mathcal{O}(n \log \log n)$. From the Bertrand–Chebyshev theorem (see proof of part (1) of Theorem 5.3, page 237), there is a prime number between $\lfloor n/p \rfloor$ and $2 \cdot \lfloor n/p \rfloor$. Therefore, the number of iterations of the loop shown above, from $\lfloor n/p \rfloor + 1$ to at most $2 \cdot \lfloor n/p \rfloor - 1$, is at most $\lfloor n/p \rfloor$. Variable p is a prime between 2 and $\sqrt{n}$. So, the total number of iterations of the loop over the entire program execution is at most $(+p : 2 \leq p \leq \sqrt{n}, \text{prime}(p) : \lfloor n/p \rfloor) \leq n \cdot (+p : 2 \leq p \leq \sqrt{n}, \text{prime}(p) : 1/p)$. Sum of reciprocals of primes up to $\sqrt{n}$ is $\mathcal{O}(\log \log \sqrt{n})$, so the given sum is $\mathcal{O}(n \log \log n)$. In order to get a linear time bound, I adopt a different procedure, recompute $\text{pred}(q)$, as described next.

Recomputing $\text{pred}(q)$ Theorem 5.3 (page 237) shows that given invariant I there is a unique q in P for which J holds. So, there is a unique $\text{pred}(q)$ as well, though $\text{pred}(q)$ may be 1, so it may not be in P .

First, observe that $\text{pred}(q) = 1$ iff $q = 2$, so $\text{pred}(q) \in P \equiv q > 2$. For example, with $p, n = 2, 2$ or $p, n = 2, 3$, $p \cdot \text{pred}(q) \leq n < p \cdot q$ is satisfied with $q = 2$ and $\text{pred}(q) = 1$.

Introduce variable s for $\text{pred}(q)$ whenever q is in P . The property of s that I exploit is that it never increases. To see this, let p, q, s and P denote the values of the corresponding variables at the start of an iteration and $p' = \text{succ}(p)$ and s' the corresponding values at the end of the iteration after J has been established.

$$\begin{aligned}
& J \\
\Rightarrow & \{\text{definition of } J\} \\
& p' \cdot s' \leq n \\
\Rightarrow & \{p' = \text{succ}(p) \text{ and } \text{succ}(p) > p\} \\
& p \cdot s' \leq n \\
\Rightarrow & \{p \cdot q > n \text{ from } J. \text{ so, } s' < q\} \\
& s' \leq s \vee s < s' < q \\
\Rightarrow & \{s < s' < q \text{ implies } s' \notin P, \text{ from } s = \text{pred}(q) \text{ in } P\} \\
& s' \leq s \vee s' \notin P \\
\Rightarrow & \{\text{from Theorem 5.3, } 1 \leq s' \leq n. \text{ So, } s' \notin P \Rightarrow s' \leq s \vee s' = 1\} \\
& s' \leq s \vee s' = 1 \\
\Rightarrow & \{\text{from loop iteration condition } p < q, \text{ so } q > 2\} \\
& (s' \leq s \vee s' = 1), q > 2 \\
\Rightarrow & \{q > 2 \wedge s = \text{pred}(q) \text{ implies } s \in P, \text{ so } s \geq 2\} \\
& s' \leq s
\end{aligned}$$

Skeleton of the final sieve program

```

initialize  $P, p, q$ ;
 $s := \lfloor n/2 \rfloor$ ;
while  $p < q$  do
  remove all multiples of  $p$  from  $P$ ;
  {  $P$  is modified }
   $p := \text{succ}(p)$ ;
  while  $\neg(s \in P \cup \{1\} \wedge p \cdot s \leq n)$  do
     $s := s - 1$ 
  enddo ;
  {  $(s \in P \cup \{1\} \wedge p \cdot s \leq n)$  }
   $q := \text{succ}(s)$ 
enddo

```

Figure 5.19

This observation suggests that the numbers may be scanned in decreasing order starting at s to find the first value that satisfies $s \in P \cup \{1\} \wedge p \cdot s \leq n$. I show a skeleton of the whole program in Figure 5.19 that incorporates this strategy.

Exercise 12 (page 261) shows that the two conjuncts in $(s \in P \cup \{1\} \wedge p \cdot s \leq n)$ can be established in sequence, first $s \in P \cup \{1\}$ using a linear search as shown above, next $p \cdot s \leq n$ by searching along the chain of predecessors using *pred*.

The complete sieve program Putting the various pieces from the previous parts, we get the complete sieve program in Figure 5.20:

Complete Eratosthenes-Sieve

```

{ initialize }
 $P := \{x \mid 2 \leq x \leq n\};$ 
 $p := 2;$ 
 $q := \lfloor n/2 \rfloor + 1;$ 
 $s := \lfloor n/2 \rfloor;$ 

while  $p < q$  do
{ remove multiples of  $p$  }
   $r := q;$ 
  while  $p \neq r$  do
     $r := \text{pred}(r);$ 
     $P := P - \{p \cdot r\}$ 
  enddo ;

{ update  $p$  and reestablish  $J$  }
   $p := \text{succ}(p);$ 
  while  $\neg(s \in P \cup \{1\} \wedge p \cdot s \leq n)$  do
     $s := s - 1$ 
  enddo ;
   $q := \text{succ}(s)$ 
enddo
{  $P = \{x \mid 2 \leq x \leq n, \text{prime}(x)\}$  }

```

Figure 5.20

5.7.5 Discussion

Data structure for P The data structure for P has to support removal of arbitrary elements from P , application of *succ* and *pred*, and test for membership. Assign an array to P , indexed from 1 to $n+1$. Entry $A[i]$ is the tuple $(in(i), pred(i), succ(i))$, where $in(i)$ is a boolean, the value of $i \in P$, $pred(i)$ and $succ(i)$ are the corresponding values in P provided $i \in P$. Thus, the elements of P are doubly linked and also accessible by index. Each elementary operation can be performed in unit time. See [Misra \[1979\]](#) for a more careful choice of data structure that takes the total number of bits of storage and array accessing time into account.

Running time estimation I prove that every loop executes at most n iterations during the execution of the program; so no test or command in a loop body is executed more than n times, so the program's running time is $\mathcal{O}(n)$. The outermost loop is executed as long as $p < q$, where p assumes new values in each iteration up to n . The loop for removal of multiples of p removes a composite in each iteration, each composite is removed exactly once, and there are at most n composites; so, its iteration count is at most n . The loop for reestablishment of J decreases s in each iteration and s can be decreased at most n times.

5.8 Stable Matching

The problem of stable matching, also known as stable marriage, is best described in the following social scenario. There is a group consisting of equal number of men and women who are willing to marry any one of the other gender in the group, though each one has a preference order for members of the other gender. The goal is to find a matching so that every person is married to a person of the other gender, and there is no pair of a man and woman, unmarried to each other, who prefer each other to their own mates. Such a matching is known as *stable matching*.² A solution for this problem was first proposed in [Gale and Shapley \[1962\]](#).

The problem is important in mathematics, computer science and economics. A well-known application is matching medical interns with hospitals. It is a stable matching problem if each hospital accepts just one intern. But the more general case, which would amount to a woman marrying several men (or vice versa), has also been studied and applied in practice. Applications include assigning jobs to machines in a machine shop where neither the jobs nor the machines are alike, but

2. I would have preferred a different version of the problem avoiding gender, though it is easier to explain the solution in social terms using men/women.

each machine can handle any of the jobs, and assigning servers to users for video downloads and interactive video meetings.

Outline of the solution There is always a stable matching. The solution procedure is actually quite simple. At any point some subset of persons are engaged, and we have the invariant that the matching is stable over this subset. If everyone is engaged, the matching is complete and stable. Otherwise, there is an unengaged man. He proposes to the woman of his highest preference to whom he has never proposed. If the proposal is the best one the woman has received so far, she accepts the proposal breaking off any earlier engagement. It can be shown that this action preserves the invariant. This step is repeated until everyone is engaged, which is then a stable matching. I formalize the problem and prove the correctness of this solution next.

5.8.1 Formal Description of the Problem and an Algorithm

Henceforth, m denotes a man, w a woman, and x and y persons of either gender. Write $a >_x b$ to mean that x prefers a to b ; here $>_x$ is a strict total order, a and b are of the same gender and x of the other gender. I also say that a has a higher *rank* than b at x , or x ranks a higher than b , if $a >_x b$.

The definition of *stable matching* is: (1) every person is engaged, and (2) given that the pairs (m, w) and (m', w') are engaged, it is not the case that m ranks w' higher than w and w' ranks m higher than m' , that is, $\neg(w' >_m w \wedge m >_{w'} m')$.

I show the abstract algorithm in Figure 5.21.

Abstract algorithm The state of the computation is given by two variables, *prop* and *engt*, that are sets of pairs of the form (m, w) . If (m, w) is in *prop*, then m has proposed to w at some time, and if (m, w) is in *engt*, then m is engaged to w .

Introduce the following derived variables to simplify the algorithm description and proof. Note, in particular, that I introduce $w.engt$, a *set* of men to whom a woman w is engaged, though this set has at most one element at all times. Using a set eliminates a separate case analysis for unengaged w .

1. *uneng* is the set of unengaged men, that is,

$$uneng = \{m \mid (\forall w :: (m, w) \notin engt)\}.$$
2. $m.prop$, the set of women to whom m has proposed, that is,

$$m.prop = \{w \mid (m, w) \in prop\}.$$
3. $w.engt$, the set of men to whom w is engaged, that is,

$$w.engt = \{m \mid (m, w) \in engt\}.$$

Abstract algorithm for stable matching

```

prop, engt := {}, {};
while uneng ≠ {} do
  m :∈ uneng; { m is any unengaged man. }
  { m chooses the woman of the highest rank to whom he has never proposed. }
  w :∈ { u | u ∉ m.prop, (∀ v : v ∉ m.prop : u ≥m v) };
  prop := prop ∪ {(m, w)}; { m proposes to w }
  { w accepts the proposal if she ranks m higher than any man in w.engt }
  if (∀ m' : m' ∈ w.engt : m >w m')
    then { if w.engt = {}, i.e. w is unengaged, the test succeeds }
      engt := engt ∪ {(m, w)} − {(m', w) | m' ∈ w.engt}
      { m is now engaged and the previous suitor, if any, of w is unengaged. }
    else skip
  endif
enddo

```

Figure 5.21**5.8.2 Correctness**

Partial correctness The structure of the proof is: prove (1) a set of invariants, (2) that an unengaged man can propose to some woman, and (3) that the matching is stable over the engaged pairs in *engt*. At termination, all persons are engaged, so *engt* defines a stable matching over all persons.

The proofs of the following invariants (Inv1, Inv2, Inv3, Inv4, Inv5) are entirely standard (see Exercise 13, page 263).

Inv1. Any engagement is preceded by a proposal:

$$engt \subseteq prop.$$

Inv2. *engt* defines a matching, that is, any two distinct pairs in *engt* are distinct in both their components:

$$(\forall (m, w), (m', w') : (m, w) \in engt, (m', w') \in engt : (m = m') \equiv (w = w')).$$

It follows from (Inv2) that a woman is engaged to at most one man.

Inv3. A woman who has ever received a proposal from any man is engaged:

$$(m, w) \in prop \Rightarrow w.engt \neq \{ \}.$$

Inv4. A man proposes in decreasing order of his ranks:

$$((m, w) \in prop, w' >_m w) \Rightarrow (m, w') \in prop.$$

Inv5. A woman is engaged to the man of her highest rank who has proposed to her:

$$((m, w') \in \text{prop}, (m', w') \in \text{engt}) \Rightarrow m' \geq_{w'} m.$$

Proposition 5.1 An unengaged man, m , can propose to some woman, that is,
 $m \in \text{uneng} \Rightarrow (\exists w :: (m, w) \notin \text{prop}).$

Proof. Prove the contrapositive.

$$\begin{aligned} & (\forall w :: (m, w) \in \text{prop}) \\ \Rightarrow & \{\text{every woman has received a proposal from } m; \text{ apply (Inv3)}\} \\ & \text{every woman is engaged} \\ \Rightarrow & \{(\text{Inv2}): \text{engt defines a matching. So, every man is engaged}\} \\ & m \notin \text{uneng} \end{aligned}$$

Proposition 5.2 engt defines a stable matching, that is,

$$\text{for all } (m, w) \text{ and } (m', w') \text{ in } \text{engt}, \neg(w' >_m w \wedge m >_{w'} m').$$

Proof. Proof is by contradiction; assume there exist $(m, w) \in \text{engt}, (m', w') \in \text{engt}$ such that $w' >_m w \wedge m >_{w'} m'$.

$$\begin{aligned} & (m, w) \in \text{engt}, w' >_m w \\ \Rightarrow & \{(\text{Inv1}): \text{engt} \subseteq \text{prop}\} \\ & (m, w) \in \text{prop}, w' >_m w \\ \Rightarrow & \{(\text{Inv4})\} \\ & (m, w') \in \text{prop} \\ \Rightarrow & \{\text{given } (m', w') \in \text{engt}; \text{ apply (Inv5)}\} \\ & m' >_{w'} m \\ \Rightarrow & \{\text{given } m >_{w'} m'\} \\ & \text{false} \end{aligned}$$

Note We proved the invariants, Inv1 through Inv5, before proving Proposition 5.1. In the absence of a proof of this proposition, we cannot claim that the execution of the command

$$w := \{u \mid u \notin m.\text{prop}, (\forall v : v \notin m.\text{prop} : u \geq_m v)\}$$

terminates. Since the proof of partial correctness is independent of termination, the proofs of the invariants are valid.

Proof of termination The size of set prop , the total number of proposals, increases in each iteration because no man proposes to the same woman twice. The maximum size of prop is N^2 , where N is the number of men (and women). Therefore, the algorithm terminates.

5.8.3 Refinement of the Algorithm

I show a refined version of the stable matching program in Figure 5.22 (page 249). Men and women, m and w , are each identified by an index between 1 and N inclusive. A numerical rank is assigned at each man to each woman and at each woman to each man; the higher the rank the more the person is preferred. Ranks range between 0 and N inclusive.

Refined program for stable matching

```

uneng := { $i \mid 1 \leq i \leq N$ }; { every man is unengaged. }
for  $w \in 1..(N+1)$  do engt[ $w$ ] := {} endfor; { every woman is unengaged. }
for  $m \in 1..(N+1)$  do lastprp[ $m$ ] := 0 endfor; { no man has proposed. }

while uneng  $\neq$  {} do
   $m \in \text{uneng}$ ; {  $m$  is any unengaged man. }
  lastprp[ $m$ ] := lastprp[ $m$ ] + 1; { the rank of the woman to whom  $m$  will propose. }
   $w := \text{mpref}[m, \text{lastprp}[m]]$ ; {  $w$  is the woman to whom  $m$  will propose. }
  if ( $\forall m' : m' \in \text{engt}[w] : \text{wpref}[w, m] > \text{wpref}[w, m']$ )
    {  $w$  is unengaged or  $w$  prefers  $m$  to her current suitor. }
    then {  $m$  is now engaged to  $w$  and the last suitor of  $w$  unengaged. }
      uneng := uneng - { $m$ }  $\cup$  engt[ $w$ ];
      engt[ $w$ ] :=  $m$  {  $m$  is the current suitor of  $w$ . }
    else skip
  endif
enddo

```

Figure 5.22

The preferences are stored in two matrices, *mpref* for men and *wpref* for women. They are arranged differently because the solution is asymmetric in men and women. Matrix entry *mpref*[m, r], for man m and rank r , is the woman whose rank is r at m . And *wpref*[w, m], for woman w and man m , is the rank of m at w .

As in the abstract solution, *uneng* is the set of unengaged men, initially all men. For each m , *lastprp*[m] is the rank at m of the last woman to whom m has proposed, initially 0. And for each w , *engt*[w] is the set of men to whom w is engaged, initially {}. The program in Figure 5.22 needs no further explanation.

Further refinements I propose a concrete data structure for *uneng*. Since the only operations on it are insertion, deletion and choosing an arbitrary element,

I propose implementing it as a doubly linked list over the indices 1 through N ; then each operation takes constant time. It is easily seen that each iteration runs in $\mathcal{O}(1)$ time and the entire algorithm in $\mathcal{O}(N^2)$.

5.9 Heavy-hitters: A Streaming Algorithm

A *streaming* algorithm makes a small, bounded number of passes over a list of data items called a stream. It processes the elements using at most logarithmic amount of extra space in the size of the list to produce an answer. Streaming algorithms are important in online applications.

5.9.1 Problem Description

I consider a problem called *heavy-hitters*. The list of data items is treated as a bag in the following discussion. The *count* of an item is the number of occurrences of that item in the bag. A *heavy-hitter* is an item whose count exceeds a given threshold. It is useful to identify the heavy-hitters in many applications. For example, a newspaper may want to know which stories have been read most frequently during a week, or a shopping website may want to identify the items that are searched most often by the consumers.

Formally, given is a bag of size N and an integer k , $2 \leq k \leq N$. A heavy-hitter is an item whose count exceeds N/k . In particular, for $k = 2$ the only heavy-hitter is an item whose count is more than half the bag size, that is, the *majority* item if one exists. The problem is to identify the heavy-hitters for a given bag and k . There may be no heavy-hitter or several heavy-hitters.

An excellent introduction to the problem, along with more recent advances, appears in [Roughgarden and Valiant \[2015\]](#); also see [Price \[2021\]](#).

A simple solution is to compute the count of each item. This is expensive in time and storage for a large bag with many distinct items, in the order of hundreds of million for the bag size and many million distinct items. Ideally, we want a one-pass streaming algorithm that examines each item exactly once and records a summary of the examined items in a (small) data structure. It computes the final result from this data structure at the end. No such algorithm exists for this problem. I show how to solve the problem using two passes over the data, and an approximate version of the problem in one pass.

The first algorithm described here, due to [Misra and Gries \[1982\]](#), was the first one discovered for this problem. It scans the items of the bag in two passes, taking $\Theta(n \log k)$ time. It requires additional storage for k items and their associated counts. The algorithm can be adapted for computing approximate heavy-hitters in one pass, as described later in this section.

There are two special cases of the problem that are worth noting. Setting k to the bag size yields all items that occur more than once, that is, have duplicates. The heavy-hitters algorithm is not particularly efficient at solving this problem because it takes $\Theta(n \log n)$ time, same as sorting the bag. The algorithm works well for small values of k . In particular, it is efficient at finding the majority item, setting k to 2, in $\Theta(n)$ time.

5.9.2 An Abstract Algorithm and its Refinement

Notation Denote the set of heavy-hitters of bag A by $hh(A)$, where parameter k is implicit. By the definition of heavy-hitters, $hh(A)$ has fewer than k elements, and it may be empty. Henceforth, B is the original bag of items, so $hh(B)$ is the set of its heavy-hitters. Extend set operations and notations to apply to bags. Write $count_A(v)$ for the number of occurrences of item v in bag A ; I drop the subscript when it is understood from the context.

Proposition 5.3 Let A' be obtained from A after removing k distinct items from A . Then all heavy-hitters of A are heavy-hitters of A' , that is, $hh(A) \subseteq hh(A')$.

Proof. Consider any heavy-hitter v of A , so $count_A(v) > |A|/k$, by definition. At most one occurrence of v has been removed in A' because the removed items are distinct. Therefore, $count_{A'}(v) > |A|/k - 1 = (|A| - k)/k = |A'|/k$. So, v is a heavy-hitter of A' , that is, every heavy-hitter of A is a heavy-hitter of A' . ■

It is important to note that $hh(A) = hh(A')$ does not necessarily hold, that is, not every heavy-hitter of A' is a heavy-hitter of A . To see this, consider $A = \{1, 2, 3\}$ and $k = 2$. Remove two distinct items, 1 and 2, from A to get $A' = \{3\}$. And $hh(A) = \{\}$ whereas $hh(A') = \{3\}$.

5.9.2.1 An Abstract Algorithm

Procedure call $remove(k, A)$ modifies A by removing k distinct items from A ; it is called only if there are at least k distinct items in A . Proposition 5.3 justifies the following specification:

$\{hh(A) = H\} \text{ remove}(k, A) \{H \subseteq hh(A)\}$, where H is an auxiliary variable.

An abstract algorithm, based on this proposition, is as follows: continue removing k distinct items from bag B as long as possible. The algorithm terminates because items cannot be removed indefinitely. Upon termination of the algorithm, we are left with a *reduced* bag R , where $hh(B) \subseteq hh(R)$. Computing $hh(R)$, I show, is a much easier task than computing $hh(B)$ directly. Each item in $hh(R)$ is a *candidate* for being a heavy-hitter of B , though it may not actually be a heavy-hitter. So, use a

second pass to count the number of occurrences in B of each candidate to identify the heavy-hitters of B .

Removing k distinct items from bag B proceeds by scanning one item of B at a time. The algorithm keeps two bags: (1) bag b is the unscanned part of B which is initially B , and (2) bag R that contains a set of candidates from the scanned items. Recall that the command $x \in b$ used in the algorithm assigns an item from non-empty bag b to x .

Correctness The program in Figure 5.23 terminates, that is, the loop terminates, because each iteration removes a single item from b and no item is ever added to b , so eventually b becomes empty. Use $|b|$ as the metric for termination.

Heavy-hitters

```

 $b, R := B, \{\};$ 
while  $b \neq \{\}$  do
   $\{b \neq \{\}\} \quad x \in b \quad \{x \in b\};$ 
   $b, R := b - \{x\}, R \cup \{x\};$ 
  if  $R$  has  $k$  distinct items
    then  $\text{remove}(k, R) \quad \{ \text{Remove } k \text{ distinct items from } R \}$ 
    else  $\{ R \text{ has fewer than } k \text{ distinct items} \} \text{ skip}$ 
  endif
enddo
 $\{ hh(B) \subseteq hh(R) \}$ 
 $\{ hh(B) = \{x \mid x \in hh(R), \text{count}_B(x) > |B|/k \} \}$ 

```

Figure 5.23

The main part of the partial correctness argument is the invariant,

$$I :: hh(B) \subseteq hh(b \cup R)$$

I sketch the proof of the invariant. The reader may derive the verification conditions and formalize the proof. Following the initialization $b, R = B, \{\}$, so I holds. In the **then** clause in the loop, observe that removing k distinct items from R also removes k distinct items from $(b \cup R)$. Using an auxiliary variable H for the value of $hh(b \cup R)$ prior to the execution of the **then** clause:

$$\begin{aligned}
 &\{hh(B) \subseteq hh(b \cup R), H = hh(b \cup R), R \text{ has } k \text{ distinct items}\} \\
 &\quad \text{remove}(k, R) \text{ \{i.e. } \text{remove}(k, b \cup R)\}} \\
 &\{hh(B) \subseteq H \subseteq hh(b \cup R)\} \\
 &\{hh(B) \subseteq hh(b \cup R)\}
 \end{aligned}$$

For the **else** clause in the loop, $b \cup R$ does not change, so I is preserved.

At the termination of the loop, from $I \wedge b = \{\}$, we have $hh(B) \subseteq hh(R)$. The postconditions of the loop are easy to see.

5.9.2.2 Refinement

Two variables, bags b and R , have been introduced in the given program. Since the items have to be removed one at a time from b in no particular order, b can be implemented by almost any data structure efficiently. Bag R can be stored as a set of tuples (v, c) , where v is an item value and $c = count_R(v)$, $c > 0$. Introduce ndv for the number of distinct items, that is, the number of tuples in R . It is initially set to zero and updated appropriately for insertion and deletion of items. The reader can show that $ndv \leq k$ is an invariant.

Insertion of an item into R , the assignment $R := R \cup \{x\}$, is refined by:

```

if  $(x, c) \in R$ 
  then  $c := c + 1$ 
  else “insert  $(x, 1)$  into  $R$ ”;  $ndv := ndv + 1$ 
endif

```

The test “**if** R has k distinct items” is simply “**if** $ndv = k$ ”. Removal of k distinct items requires scanning each tuple of R , modifying its count, and modifying ndv if the count becomes 0. Technically, a **for** loop cannot modify a set it is scanning; so, we use a proxy R' for R into which the modified tuples are added, and R is assigned R' at the end of the loop.

```

 $R' := \{\}$ ;
for  $(v, c) \in R$  do
  if  $c = 1$ 
    then { remove  $(v, c)$  from  $R$ ; i.e. do not add it to  $R'$  }  $ndv := ndv - 1$ 
    else {  $c > 1$  }  $R' := R' \cup \{(v, c - 1)\}$ 
  endif
endfor;
 $R := R'$ 

```

The heavy-hitters of R are then easily computed: for each (c, v) in R , $count_R(v) = c$. Therefore, v is a heavy-hitter of R if $c > |R|/k$; so, a single scan over R identifies its heavy-hitters.

Further refinement: Data structure for R The main operations that have to be implemented on R are: (1) inserting an element and (2) removing k distinct items. The original heavy-hitter algorithm in Misra and Gries [1982] proposed a data structure, AVL-tree, that gives a guarantee of $\mathcal{O}(\log k)$ time for each iteration, so the total running time is $\mathcal{O}(n \log k)$. In practice, it would be more efficient to store R in a hash

table so that search and insertion are performed efficiently. Additionally, for efficient scanning of the elements and deletion, impose a doubly linked list structure on the tuples of R .

5.9.3 One-pass Algorithm for Approximate Heavy-hitters

An ideal streaming algorithm examines each data item exactly (or at most) once. What can we infer from the reduced bag R at the end of the first pass? Surprisingly, the algorithm can identify all heavy-hitters plus some approximate heavy-hitters. Specifically, given two parameters p and e , $p > e$, define v as a heavy-hitter if $\text{count}_B(v) \geq p \cdot N$ and u as an approximate heavy-hitter if $\text{count}_B(u) \geq (p - e) \cdot N$; so every heavy-hitter is also an approximate heavy-hitter. For example, with $N = 10^7$, $p = 10^{-4}$ and $e = 10^{-5}$, each heavy-hitter occurs at least a thousand times in B and approximate heavy-hitters at least 900 times. See Charikar et al. [2004] for general techniques to solve this and related streaming problems.

Let hh_m be the set of elements that occur at least m times in B . The one-pass algorithm computes a set of approximate heavy-hitters, approx , that satisfies:

$$hh_{(p \cdot N)} \subseteq \text{approx} \subseteq hh_{(p-e) \cdot N}. \quad (\text{Approx-spec})$$

So, approx contains *all* heavy-hitters, *some* approximate heavy-hitters, and *no* non-approximate heavy-hitter.

The strategy for computing approx is: (1) set $k = 1/e$ and run the first pass of the heavy-hitters algorithm so that every item that occurs at least $e \cdot N$ times in B is in the reduced bag R at the end of the pass, and (2) then compute

$$\text{approx} = \{v \mid (v, c) \in R, c \geq (p - e) \cdot N\}.$$

Proposition 5.4 Let $C = \text{count}_B(v)$, and R the reduced bag.

1. If $(v, c) \in R$ then $C \geq c \geq (C - e \cdot N)$.
2. If $(C \geq e \cdot N)$ then $(v, c) \in R$, for some c .

Proof of (1): $C = \text{count}_B(v) \geq \text{count}_R(v) = c$. For $c \geq (C - e \cdot N)$, note that in the entire algorithm N items are added to R , and each execution of $\text{remove}(k, R)$ removes k items from R . So, $\text{remove}(k, R)$ is executed at most $N/k = e \cdot N$ times. Each execution of $\text{remove}(k, R)$ removes at most one instance of any v ; so R has at least $(C - e \cdot N)$ instances of v , or $c \geq (C - e \cdot N)$.

Proof of (2): The first pass of the algorithm identifies all items that occur at least $e \cdot N$ times in B . So, for any such item v , $(v, c) \in R$ for some c . ■

Theorem 5.5 Approximate heavy-hitters

Set approx meets its specification (Approx-spec) where

$$\text{approx} = \{v \mid (v, c) \in R, c \geq (p - e) \cdot N\}.$$

Proof. Below, C is the count of an arbitrary element v of B . We have to show

Ahh1. $hh_{p \cdot N} \subseteq approx$, that is, $(C \geq p \cdot N) \Rightarrow (v \in approx)$.

Ahh2. $approx \subseteq hh_{(p-e) \cdot N}$, that is, $(v \in approx) \Rightarrow (C \geq (p - e) \cdot N)$.

• Proof of (Ahh1)

$$\begin{aligned}
 & C \geq p \cdot N \\
 \Rightarrow & \{p > e, \text{ so } C \geq e \cdot N. \text{ Use Proposition 5.4, part (2)}\} \\
 & C \geq p \cdot N, (v, c) \in R, \text{ for some } c \\
 \Rightarrow & \{\text{Use Proposition 5.4, part (1), on the second conjunct}\} \\
 & C \geq p \cdot N, c \geq (C - e \cdot N), (v, c) \in R \\
 \Rightarrow & \{\text{algebra}\} \\
 & c \geq (p - e) \cdot N, (v, c) \in R \\
 \Rightarrow & \{approx = \{v \mid (v, c) \in R, c \geq (p - e) \cdot N\}\} \\
 & v \in approx
 \end{aligned}$$

• Proof of (Ahh2)

$$\begin{aligned}
 & v \in approx \\
 \Rightarrow & \{approx = \{v \mid (v, c) \in R, c \geq (p - e) \cdot N\}\} \\
 & (v, c) \in R, c \geq (p - e) \cdot N \\
 \Rightarrow & \{C \geq c, \text{ from Proposition 5.4, part (1)}\} \\
 & C \geq (p - e) \cdot N
 \end{aligned}$$

■

Most-common items A variation of the heavy-hitters problem is to find the most common m items in a bag in order of their counts. A newspaper may want to identify the most popular news items of the week in order, a store its five most commonly selling items, and a journal its ten most cited papers. The 2-pass heavy-hitters algorithm can be modified to solve this problem. Start with a reasonable estimate of k so that the most common items are among the heavy-hitters. For example, a newspaper could use historical data to suggest an estimate of k .

At the end of the second pass the number of occurrences of the heavy-hitters have been computed; so, the heavy-hitters can be ranked, and the most common m items in order can be identified. In particular, the heavy-hitter that occurs most often in the bag is the most common element in the bag.

5.9.4 The Majority Element of a Bag

The inspiration for the heavy-hitters problem came from an algorithm due to [Boyer and Moore \[1991\]](#) that identifies the majority element of a bag, if one exists. This is the heavy-hitters problem with $k = 2$. Show that the reduced bag R has at most

one pair (v, c) at any moment, thus avoiding the complexity of the data structures required for the general heavy-hitters solution.

5.10 Exercises

Many of these exercises are not specifically related to the topics covered in this chapter. They ask you to develop and prove algorithms for moderately hard problems. In most cases, I supply hints and solutions.

1. Given is a matrix of numbers. A *sweep* operation on a row replaces all elements of that row by the average value of those elements; similarly, a sweep on a column. The following algorithm is claimed to replace every element of the matrix by the average of the matrix elements:

sweep all the rows in arbitrary order;
sweep all the columns in arbitrary order

Write a program and prove its correctness. Treat sweep as a primitive operation; do not write its code but specify its pre- and postconditions.

Hint Observe that after the sweep of the rows, all the columns are identical.

2. The minimum *exudant*, *mex*, of a set of natural numbers is the smallest natural number not in the set. Thus, *mex* of $\{1, 2\}$ is 0, of $\{0, 2\}$ is 1, and of $\{0, 1\}$ is 2.

Let $A[0..N]$ be an array of distinct natural numbers in sorted order, so that $A[i] < A[i + 1]$ for all i , $0 \leq i < N - 1$. Write $\text{mex}(A)$ for *mex* of the set corresponding to A .

- (a) Suppose array A is available only as a stream of elements in order of their indices, and you can process each element only once. Devise an algorithm to compute $\text{mex}(A)$ and prove its correctness.

Hint Show that $\text{mex}(A) = j$ for the smallest index j such that $A[j] > j$, and $\text{mex}(A) = N$ if there is no such index.

- (b) Suppose the elements of A can be accessed in random order like any array. Devise an efficient algorithm for computing $\text{mex}(A)$ and prove its correctness.

Hint Use binary search. Note that if $A[j] > j$, then $A[j'] > j'$ for any j' , $j' \geq j$.

3. This is a variation of the previous exercise. Let $A[0..N-1]$ be an array of distinct natural numbers that is unsorted. Devise an efficient algorithm for computing $\text{mex}(A)$, and prove its correctness.

Hint Introduce a boolean array $B[0..N]$, longer than A by 1. Initially, all elements of B are *false*. Scan the elements of A in any order, and when $A[i]$ is scanned, execute **if** $A[i] < N$ **then** $B[A[i]] := \text{true}$ **else skip**. Thus, $B[N]$ remains *false*. Then $\text{mex}(A)$ is the smallest j such that $\neg B[j]$.

4. It is required to find the integer square root of a given natural number k . Specifically, compute a natural number s satisfying $s^2 \leq k \wedge (s+1)^2 > k$. In order to get an invariant I from this predicate, replace 1 by a new variable, b . Also, add the requirement that b be a power of 2, so that

$$I :: s^2 \leq k \wedge (s+b)^2 > k \wedge (\exists i : i \geq 0 : b = 2^i)$$

Develop a program using this invariant, and prove its correctness.

Next, solve the same problem using binary search (see Section 5.1). Use another variable t and the invariant $s^2 \leq k \wedge t^2 > k$, and establish the termination condition $t = s + 1$.

5. You are given a positive integer n . Write a program to find all pairs of positive integers a and b , $a \leq b$, such that $n = a^3 + b^3$. Note that there may be zero, one or more than one such pair for a given n . For instance, since $1^3 + 1^3 = 2$ and $1^3 + 2^3 = 9$, there is no such pair for any n , $2 < n < 9$. And, 9 has exactly one pair corresponding to it. The smallest positive integer that has two pairs corresponding to it is 1729: $1729 = 10^3 + 9^3 = 12^3 + 1^3$.

Hint Define matrix A where $A[i,j] = i^3 + j^3$, $1 \leq i \leq n^{1/3}/2$, $1 \leq j \leq n^{1/3}$. Search for n in this matrix using saddleback search.

6. Consider the saddleback search problem of Section 5.2 (page 205) when the number of rows and columns of the matrix are vastly different. Develop a better search strategy using binary search.

Hint Let us do a binary search on the middle row for a given value k . If we find it, we are done. Otherwise, we have two adjacent numbers y and z in the row such that $y < k < z$. Eliminate the submatrix of which y is the bottom rightmost corner because all numbers in it are smaller than k . Similarly, eliminate the submatrix of which z is the top leftmost corner because all numbers in it are greater than k . Search the remaining two submatrices using a similar strategy. Apply saddleback search on any submatrix that has comparable number of rows and columns [see Bird 2006].

7. (Dutch national flag) Edsger Dijkstra posed and solved this problem in a chapter in Dijkstra [1976]. Given an array A of numbers and a pivot element

p , permute the elements of A such that A is a sequence of three segments, L, C and R, where every element (1) of L is strictly less than p , (2) of C equal to p , and (3) of R greater than p (see Table 5.5). Use a variation of the partition procedure of Section 5.4.3 (page 212) to achieve this partition.

Table 5.5 Partitioning in Dutch national flag

L	C	R
<	= p	>

Hint As an invariant, A is partitioned into four segments: L, C, U and R. Segments L, C and R have the same meaning as described in the problem, and segment U is unscanned (see Table 5.6). In each step, reduce the size of unscanned. Eventually, unscanned is empty, so the desired partitioning has been achieved.

Table 5.6 Unscanned segment in partitioning in Dutch national flag

L	C	U	R
<	= p	?	>

Solution The segments L, C, U and R correspond to the intervals $0..i$, $i..j$, $j..k$ and $k..n$, respectively, where A is $A[0..n]$. Initially, all segments except U are empty. In each step, the leftmost element of U, $A[j]$, is compared against p and moved to one of the other segments so that the length of U decreases. Annotate and prove the following program.

```

 $i := 0; j := 0; k := n;$ 
while  $j < k$  do
  if  $A[j] < p$ 
  then
     $A[i], A[j] := A[j], A[i];$ 
     $i := i + 1; j := j + 1$ 
  else
    if  $A[j] > p$ 
    then  $A[j], A[k-1] := A[k-1], A[j]; k := k - 1$ 
    else  $j := j + 1$ 
    endif
  endif
enddo

```

8. (Identifying Pareto points) In a given set of points in the Euclidean plane, a point *dominates* another if both of its coordinates are greater than or equal to

the corresponding coordinates of the latter, that is, for distinct points (a, b) and (a', b') , (a, b) dominates (a', b') if $a \geq a' \wedge b \geq b'$. A *pareto* point is one that is dominated by no other point in the set. Design an algorithm to identify all pareto points of a given set.

Hint Let us call (a, b) *higher* than (a', b') if (a, b) is lexicographically greater, that is, $(a > a') \vee (a = a' \wedge b > b')$. Scan the points of the set from the highest to the lowest in lexicographic order. A scanned point (a, b) is pareto if b is greater than the largest y coordinate scanned so far.

Below, S is the given set of points and $y_{\max}$ the largest y coordinate scanned so far.

```

 $y_{\max} := -\infty;$ 
while  $S \neq \{\}$  do
  remove the highest point  $(a, b)$  from  $S$ ;
  if  $b > y_{\max}$ 
    then “identify  $(a, b)$  as a pareto point”;
     $b := y_{\max}$ 
  else skip
  endif
enddo

```

Augment this program with additional variables to state its specification. Prove that all and only pareto points are identified. First, prove the following result: if (a, b) dominates (a', b') , then (a, b) is higher than (a', b') .

Solution Let S_0 be the given set of points and $S' = S_0 - S$ be the set of scanned points; initially, $S' = \{\}$, and finally, $S' = S_0$. Let P be the set of points that have been identified as pareto points and $\text{pareto}(A)$ the set of pareto points in any set A according to the definition. Prove the invariant:

$$P = \text{pareto}(S') \wedge (\forall x, y: (x, y) \in S': y_{\max} \geq y).$$

9. (Generating a sequence without duplicates) Given function $f, f : S \rightarrow S$, sequence X is generated by starting with some value x_1 from S and applying f repeatedly: $X = \langle x_1, x_2, \dots \rangle$, where $x_{i+1} = f(x_i)$ for all i . For example, pseudo-random numbers are generated by starting from seed x_1 and repeatedly applying a randomizing function. In such applications, it is essential to avoid generating duplicates; so we seek a method to output a prefix of X without duplicates that is as long as possible. Duplicates are inevitable if S is a finite set.

The following method, attributed to Floyd in Knuth [2014], solves the problem at the expense of a few words of extra storage and about triple the

computation time. Compute x_i and $x_{2 \cdot i}$ for increasing values of i starting at $i = 1$; output x_i if $x_i \neq x_{2 \cdot i}$.

Write a program for this method and prove its correctness.

Solution First, I develop the underlying theory. Given that $f : S \rightarrow S$ and S is a finite set, there is a duplicate in some prefix of X . That is, for some m and p , where $m > 0$ and $p > 0$, $x_m = x_{m+p}$. Using Exercise 12 of Chapter 3 (page 128) deduce that

$$\text{for all } i \text{ and } k, \text{ where } i \geq m \text{ and } k > 0, x_i = x_{i+k \cdot p}. \quad (1)$$

Consider the interval $m..(m + p)$. There are p consecutive numbers in this interval. Using the pigeonhole principle, Section 2.2.7 (page 23), there is some i , $i \in m..(m + p)$, such that $i \equiv 0 \pmod{p}$, or $i = k \cdot p$ for some k , $k > 0$. Then

$$\begin{aligned} & x_i \\ = & \{i \geq m \text{ and } k > 0. \text{ Apply (1)}\} \\ & x_{i+k \cdot p} \\ = & \{i = k \cdot p\} \\ & x_{2 \cdot i} \end{aligned}$$

Call x_j a *doubler* if $x_j = x_{2 \cdot j}$. I have just shown that any prefix of X that contains a duplicate also contains a doubler. Conversely, if $1..j$ has no doubler, it has no duplicate, so it is safe to output x_i for all i in this interval. Encode the method in the following program using i as an auxiliary variable.

```

r, s, i := x1, f(x1), 1;
while r ≠ s do
  output r;
  r, s, i := f(r), f(f(s)), i + 1
enddo

```

Postulate the invariant:

$$i \geq 1, r = x_i, s = x_{2 \cdot i}, (\forall j : 1..i : x_j \neq x_{2 \cdot j}).$$

10. Given strings x and y of equal length, determine if x is a rotation of y . Solve the problem through string matching.

Solution Determine if x occurs as a pattern in yy .

11. Show that *core* function of Section 5.6.2.1 (page 225) is monotonic, that is, $U \preceq V \Rightarrow c(U) \preceq c(V)$.

Solution

$$\begin{aligned}
& U \preceq V \\
\Rightarrow & \{ \text{from Theorem 5.2, } U = c^i(V), \text{ for some } i, i \geq 0 \} \\
& c(U) = c^{i+1}(V), i \geq 0 \\
\Rightarrow & \{ c^{i+1}(V) = c^i(c(V)). \text{ From Theorem 5.1 (page 226), } c^i(c(V)) \preceq c(V) \} \\
& c(U) \preceq c(V)
\end{aligned}$$

12. The sieve program in Figure 5.20 (page 244) locates a new value for s by scanning the values in decreasing order starting at its current value, to find the first value that satisfies $(s \in P \cup \{1\} \wedge p \cdot s \leq n)$. The following program fragment first scans the values to locate s such that $s \in P$ holds, then (since $s \in P$) applies pred repeatedly until $p \cdot s \leq n$ holds. Prove the correctness of this strategy.

```

while  $s \notin P$  do
   $s := s - 1$ 
enddo ;
while  $p \cdot s > n$  do
   $s := \text{pred}(s)$ 
enddo ;
 $\{ (s \in P \vee s = 1), p \cdot s \leq n < p \cdot \text{succ}(s) \}$ 
 $q := \text{succ}(s)$ 

```

Hint First loop: Let p_0 be the value of p before $p := \text{succ}(p)$ was executed. Then $p_0 \leq s$ is an invariant of the first loop. Also, $p_0 \in P$; so the loop terminates. The postcondition of the loop is $s \in P$, and none of the values greater than s are candidates for $\text{pred}(q)$.

Second loop: The second loop terminates because $p \cdot 1 \leq n$, so 1 is a lower bound for s . At termination, $p \cdot s \leq n$. Also formulate an invariant that no higher value of s satisfies $p \cdot s \leq n$.

Solution Define $\text{next}(s)$ for any $s, 2 \leq s \leq n$, to be the next larger value than s in P , or $n + 1$ if there is no such value. If $s \in P$, then $\text{next}(s) = \text{succ}(s)$. Define $L, L :: \text{next}(s) \geq q$.

I show the skeleton of the whole program in Figure 5.24 with annotation that is pertinent only for the recomputation of s .

Eratosthenes-Sieve2

```

initialize  $P, p, q$ ;
 $s := \lfloor n/2 \rfloor$ ;

{  $n < p \cdot q, p \in P, (p \leq s < q \vee p = q), L$  };
while  $p < q$  do
  {  $p \leq s, p \in P, n < p \cdot q, L$  }
  remove all multiples of  $p$ ;

  {  $p_0 = p, p \leq s, p \in P, n < p \cdot q, L$  }
   $p := \text{succ}(p)$ ;

  {  $p_0 \leq s, p_0 \in P, n < p \cdot q, L$  }
  while  $s \notin P$  do
    {  $p_0 < s, p_0 \in P, n < p \cdot q, L, s \notin P$  }
     $s := s - 1$ 
  {  $p_0 \leq s, p_0 \in P, n < p \cdot q, L$  }
  enddo ;
  {  $s \in P, n < p \cdot q, L$  }

  {  $s \in P, n < p \cdot q, q \leq \text{succ}(s)$  }
  {  $(s \in P \vee s = 1), n < p \cdot \text{succ}(s)$  }
  while  $p \cdot s > n$  do
    {  $s \in P, p \cdot s > n$  }
     $s := \text{pred}(s)$ 
    {  $(s \in P \vee s = 1), n < p \cdot \text{succ}(s)$  }
  enddo ;
  {  $(s \in P \vee s = 1), p \cdot s \leq n < p \cdot \text{succ}(s)$  }

   $q := \text{succ}(s)$ 
  {  $J :: q \in P \wedge p \cdot \text{pred}(q) \leq n < p \cdot q$  }
enddo

```

Figure 5.24

13. Prove the following invariants in the given sequence for the abstract stable matching algorithm shown in Figure 5.21 (page 247).

Inv1. Any engagement is preceded by a proposal:

$$engt \subseteq prop.$$

Inv2. $engt$ defines a matching, that is, any two distinct pairs in $engt$ are distinct in both their components:

$$(\forall (m, w), (m', w') : (m, w) \in engt, (m', w') \in engt : m = m' \equiv w = w').$$

It follows from (Inv2) that a woman is engaged to at most one man.

Inv3. A woman who has ever received a proposal from any man is engaged:

$$(m, w) \in prop \Rightarrow w.engt \neq \{\}.$$

Inv4. A man proposes in decreasing order of his ranks:

$$((m, w) \in prop, w' >_m w) \Rightarrow (m, w') \in prop.$$

Inv5. A woman is engaged to the man of her highest rank who has proposed to her:

$$((m, w') \in prop, (m', w') \in engt \Rightarrow m' \geq_{w'} m.$$

14. (Stable matching) Consider the following algorithm for stable matching: any man or woman who is unengaged, proposes to the best person in their ranking to whom he/she has not proposed earlier. And a person accepts a proposal if it is the best she/he has received, possibly breaking off an earlier engagement.

Show that this solution is wrong. Instead of looking for a counterexample, check where the proof breaks down, and construct a counterexample.

Hint Checking the invariants in Exercise 13, observe that (Inv4) breaks down. In Section 5.8.2 (Inv4) is used in Proposition 5.2 (page 248). So, the matching would not be stable. Construct a counterexample based on this observation.

Graph Algorithms

6.1 Introduction

I review the elementary concepts of graph theory in this chapter and describe a few basic algorithms that are extensively used in a variety of disciplines.

A graph is used to depict a finite or infinite set of objects and the relationships between pairs of those objects. I consider only finite graphs in this book. Each object is called a *node* and the relationship an *edge*. As an example, a road network may be regarded as a graph where the towns are nodes and the roads connecting the towns are edges. Often a road may have an associated label, such as its length, the time to travel over it or the number of toll booths on it. Some of the relevant questions that may then be posed are: (1) is it possible to go from one town to another, (2) what is the minimum distance or travel time between two towns, and (3) the same questions as in (2) avoiding any toll.

The objects and the relationships in a graph are drawn from a variety of fields. Almost every science and engineering discipline has major uses for graphs. In recent years, their uses in computer science have exploded with the advent of computer communication networks, social (human) networks and web graphs, and neural networks and knowledge graphs that are used in machine learning. Hardware circuit designers draw circuits (with the help of computers) to minimize edge crossings. Operating systems employ graphs to allocate resources among competing users while avoiding deadlock; here users and resources are the nodes, and an edge exists only between a user and a resource. Assignments of jobs to machines for completion within the least possible time, transporting resources from multiple sources to multiple destinations where bridges on the roads may impose constraints on the amount that can be transported, and finding the best route for a postal worker to deliver all mail in a neighborhood are problems in operations research. Even in the area of marketing, product recommendation graphs are used to offer (and often pester) users to buy specific items. Many questions in these

fields may be posed as problems in graph theory for which some of the algorithms of this chapter may be applicable.

On large graphs Graphs that arise in practice can be large, very large. Large graphs were only of theoretical interest before the advent of large-scale computers. It is claimed that in early 2020 there are over six billion pages in the World Wide Web, and these pages are connected to the other pages via hyperlinks that are similarly large; a study reports that there are over 1.5 billion hyperlinks in a subset of 200 million pages. It is estimated that there were over twenty billion internet connected devices in 2018, and the number will likely double by 2025. Any graph algorithm that will process a graph of this size has to be very efficient. Even a quadratic-time algorithm may be too slow. But these graph sizes are dwarfed by the problems that arise in biology. The brain of every living animal and insect consists of neurons, and the neurons are connected through synapses. A connectome is a graph that displays all neural connections within an organism's nervous system. A fruit fly has over 100,000 neurons, a mouse over 70 million and humans around 85 billion. A quarter of a fruit fly's neurons are connected using around 20 million synapses. The connectome of a human brain is astronomical in size. Actually, "astronomical" does not quite describe the size; our galaxy has only about 100 billion stars, relatively puny in comparison with the number of human synapses, estimated around 0.15 quadrillion.

6.2 Background

6.2.1 Directed and Undirected Graph

A finite *graph* is defined by a finite set of *nodes* and a finite set of *edges*, where each edge is a pair of nodes. Other common terms are *vertex* for a node and *arc* for an edge. A graph, typically, models a physical system in which a node denotes an object and an edge the relationship between two objects. In this chapter, I do not assign meanings to nodes and edges; I develop the theory to study the mathematical properties of graphs, independent of the domain it models. Observe that there may be multiple edges between a pair of nodes, in which case the edges cannot be described by a *set* of pairs of nodes, but by a *bag*. Multiple edges can be replaced by a single edge if the edges carry no additional information.

To regard a graph as merely a binary relation over some prescribed set is inadequate for both intuitive understanding and study of its formal properties. It is often useful to depict a graph by a picture, as shown in Figure 6.1. In Figure 6.1(a), the nodes are shown as points and the edges as arrows connecting the points, where an edge is *directed* from node u to v if there is an edge (u, v) . For this graph:

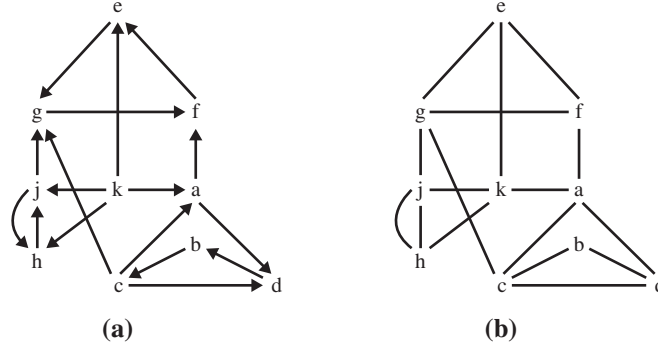


Figure 6.1 A directed and the corresponding undirected graph.

Nodes: $\{a, b, c, d, e, f, g, h, j, k\}$, and

Edges: $\{(a, d), (a, f), (b, c), (c, a), (c, d), (c, g), (d, b), (e, g),$
 $(f, e), (g, f), (h, j), (j, g), (j, h), (k, a), (k, e), (k, h), (k, j)\}$.

The *underlying undirected graph* of a directed graph is obtained by ignoring the edge directions; Figure 6.1(b) is the underlying undirected graph for Figure 6.1(a). We may regard an edge in an undirected graph to point in both directions, that is, there is an edge (u, v) iff there is edge (v, u) . In Figure 6.1(b) there are two edges between j and h corresponding to two directed edges between them; we can replace these two edges by a single edge.

Edge (u, v) is *incident* on nodes u and v . In a directed graph, (u, v) is an *outgoing* edge of u and an *incoming* edge of v . Then v is a *successor* of u and u a *predecessor* of v . Nodes u and v are *adjacent* if there is an edge between them in either direction. Dually, two edges are adjacent if they have a common node. For node a in Figure 6.1(a), the incident edges are $\{(k, a), (a, f), (c, a), (a, d)\}$, where (k, a) and (c, a) are incoming and (a, f) and (a, d) outgoing, k is a predecessor and d a successor of a . Nodes k and a and edges (k, a) and (a, d) are adjacent in Figure 6.1(a). These definitions also apply to undirected graphs except, since there is no edge direction, incoming and outgoing edges are just incident edges.

In an undirected graph, the *degree* of a node is the number of edges incident on it, so the degrees of k and a are both 4 in Figure 6.1(b). The *in-degree* of a node in a directed graph is the number of its incoming edges and *out-degree* the number of its outgoing edges. A node of in-degree 0, that is, without an incoming edge, is called a *source* and of out-degree 0, that is, without an outgoing edge, a *sink*. In Figure 6.1(a), the in-degree of k is 0 and out-degree 4; so k is a source. Both in-degree and out-degree of a are 2; so it is neither a source nor a sink. The given graph has no sink node.

Sparse and dense graph Sparse graphs don't have too many edges, dense graphs do. There is no standard mathematical definition of sparse or dense graph; what is considered sparse or dense depends on context. Call a graph of n nodes sparse if it has $\mathcal{O}(n^{1+\varepsilon})$ edges and dense if it has $\Omega(n^{2-\varepsilon})$ edges, for some small positive value of ε much smaller than 1, say 0.1. I use the terms sparse and dense informally in this chapter.

6.2.2 Paths and Cycles

In a directed or undirected graph, a *path* is a sequence of nodes $\langle u_0 u_1 \dots u_n \rangle$, $n \geq 0$, where edge (u_i, u_{i+1}) exists for each i , $0 \leq i < n$; for $n = 0$ the path is a trivial one consisting of a single node. In Figure 6.1(a), $\langle k j g f e \rangle$ and $\langle c d b \rangle$ are paths; the corresponding paths also exist in Figure 6.1(b). A *directed path* is one where the edges have directions and an *undirected path* has no edge direction. The terminology is derived from analogy with road networks where the roads can be one-way or two-way, and a route can be traveled along its roads only in the specified direction. A *simple path* is one where no node is repeated, *non-simple* if a node is repeated. The example paths $\langle k j g f e \rangle$ and $\langle c d b \rangle$ are both simple whereas $\langle k j g f e g \rangle$ is non-simple.

A *cycle* is a path whose first and last node are identical. So, traversing the sequence of edges along a cycle brings one to the starting point. In Figure 6.1(a), $\langle g f e g \rangle$ is a cycle. The given cycle is a directed one; similar definitions apply for the undirected case. The nodes of a cycle can be enumerated starting at any of its nodes, so $\langle g f e g \rangle$ can be written equivalently as $\langle f e g f \rangle$. A *simple cycle* $\langle u_0 u_1 \dots u_n \rangle$ has no repeated nodes except $u_0 = u_n$. In Figure 6.1(a), $\langle a d b c d b c a \rangle$ is a non-simple cycle.

The length of a simple path (or cycle) is the number of edges in it. For a simple cycle of n nodes, the length is n ; for a simple path of n nodes, the length is $n - 1$.

For a simple non-cyclic path $\langle u_0 u_1 \dots u_n \rangle$, so $u_0 \neq u_n$, both u_0 and u_n are *exterior* nodes and the remaining nodes are *interior* nodes. A cycle has only interior nodes.

Notation Write $u \rightarrow v$ to denote that there is a directed edge from node u to v and $u - v$ for an undirected edge. And $u \rightsquigarrow v$ for a directed path and $u \rightsquigarrow v$ for an undirected path. Add a label to an edge or a path, as in $u \xrightarrow{p} v$, $u \overset{p}{\rightsquigarrow} v$, $u \xrightarrow[p]{} v$ or $u \overset[p]{\rightsquigarrow} v$ to identify a specific edge or path. For example, if nodes u and v are on a directed cycle, there are two paths p and q , $u \xrightarrow{p} v$ and $v \xrightarrow{q} u$, between them.

The cycle $\langle g f e g \rangle$ in Figure 6.1(a) is shown as $g \rightarrow f \rightarrow e \rightarrow g$ and the corresponding cycle in Figure 6.1(b) as $g - f - e - g$. Just to show the path from g to e without enumerating the nodes in it write $g \rightsquigarrow e$ and $g \rightsquigarrow e$ in Figure 6.1(a) and Figure 6.1(b), respectively.

6.2.3 Connected Components

Undirected graph Nodes u and v are *reachable* from each other in an undirected graph if there is a path between u and v . In this case, u and v are *connected*. In Figure 6.2(a), a and b are reachable from each other and so are g and k . But a and g are not reachable from each other. Reachability is an equivalence relation in undirected graphs.

The nodes of an undirected graph are partitioned into disjoint subsets, called *components*, based on the reachability relation, so that connected nodes belong to the same component. The set of edges are also partitioned so that both incident nodes of an edge belong to the same component. The components in Figure 6.2(a) are readily discernible: $\{a, b, c\}$, $\{d, e\}$ and $\{f, g, h, k\}$. An undirected graph is *connected* if it has just one component.

Directed graph Node v is *reachable* from u in a directed graph if there is a directed path from u to v . In Figure 6.2(b), e and k are both reachable from d , but d is reachable from neither. Thus, in contrast to undirected graphs, reachability is not a symmetric relation for directed graphs.

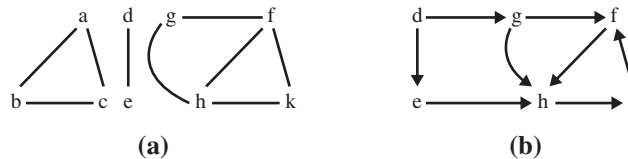


Figure 6.2 Example graphs to illustrate connected components.

Nodes u and v are *strongly connected* in a directed graph if both nodes are reachable from each other; then there is a cycle that includes both u and v . A directed graph is strongly connected (or *completely connected*) if every pair of nodes are strongly-connected. The nodes of a directed graph can be partitioned into a set of strongly connected components, as in an undirected graph. However, unlike an undirected graph, nodes u and v that are connected by a directed edge $u \rightarrow v$ may belong to different strongly connected components. The strongly connected components of the directed graph in Figure 6.2(b) are not so easy to see; they are: $\{d\}$, $\{e\}$, $\{g\}$, and $\{f, h, k\}$.

6.2.4 Labeled Graph

I have described graphs only by their structure, the nodes and the connections among them. Many applications require additional information beyond the structure. For example, a road network may include information about the distances among the towns, which can be added as a label on the corresponding edge. This

information is needed to determine a shortest path between a specified pair of towns (see Section 6.9). Finite state machines have labels, that are symbols from an alphabet, on the edges of a directed graph.

A project management tool, called PERT chart, models each task in the project as a node; if task q can start only after the completion of task p , then there is an edge from p to q . Such a graphical depiction is visually useful in assessing dependency among the tasks, most importantly that there is no cycle in the graph. The model becomes far more useful by labeling each node by the amount of time and the set of resources required to complete the associated task. The critical tasks can then be identified, the length of the project can be assessed, and resources more efficiently deployed.

6.2.5 Graph Representation

A graph can be represented in a number of different ways for processing by a computer. A particular representation for a specific application may be suitable for efficient processing. I sketch two different representations, adjacency matrix and adjacency list, that are commonly used in many applications. Other representations combine the common data structures of computer science, such as hash tables, trees and linked lists.

Linked list representation A directed graph can be represented by a set of *successor sets*, the successors of each node. The successor sets for various nodes of Figure 6.1(a) (page 267) are shown below.

$$\begin{aligned} a: \{d, f\}, b: \{c\}, c: \{a, d, g\}, d: \{b\}, e: \{g\}, f: \{e\}, g: \{f\}, \\ h: \{j\}, j: \{g, h\}, k: \{a, e, h, j\} \end{aligned}$$

A successor set is often implemented as a linked list and is then called an *adjacency list*.

Adjacency matrix representation A graph can be represented by a matrix, called its *adjacency matrix* (or the node adjacency matrix), where each row and each column is labeled with a node name and the matrix entry at (i, j) is *true* iff there is an edge from i to j . Often 0 and 1 are used in place of *false* and *true*, respectively.

The adjacency matrix for the graph of Figure 6.1(a) (page 267) is shown in Table 6.1, where T stands for *true* and F for *false*. For this matrix A , $A[i, i]$ is *false* unless there is an edge from i to i ; in the given graph there is no such edge.

The linked list representation is space-efficient compared to the adjacency matrix, especially for sparse graphs. It is equally time-efficient if all the edges have to be explored in some fashion, as we will see in traversal algorithms in Sections 6.6.1 (page 293) and 6.6.2 (page 296). Adjacency matrix representation is

Table 6.1 Adjacency matrix A for Figure 6.1

	a	b	c	d	e	f	g	h	j	k
a	F	F	F	T	F	T	F	F	F	F
b	F	F	T	F	F	F	F	F	F	F
c	T	F	F	T	F	F	T	F	F	F
d	F	T	F	F	F	F	F	F	F	F
e	F	F	F	F	F	F	T	F	F	F
f	F	F	F	F	T	F	F	F	F	F
g	F	F	F	F	F	T	F	F	F	F
h	F	F	F	F	F	F	F	F	T	F
j	F	F	F	F	F	F	T	T	F	F
k	T	F	F	F	T	F	F	T	T	F

faster for random access to the edges. When edges carry labels, such as their lengths, the matrix entries are the labels instead of truth values.

Undirected graph The adjacency matrix of an undirected graph is symmetric, that is, edge $i—j$ exists iff $j—i$ exists, so only the lower or upper triangle of the matrix needs to be stored. The linked list representation has two elements for an edge $u—v$, once in the list for u and once for v , typically for efficiency in processing.

6.2.6 Graph Manipulation: Merging, Pruning and Joining

Some of the most common operations on graphs (or sets of graphs) include adding/removing nodes and edges, merging nodes, removing a portion of a path (*pruning*) and concatenating (*joining*) two paths. Recall that a cycle is also a path, so pruning and joining also applies to cycles. The following operations on directed graphs have counterparts for undirected graphs.

6.2.6.1 Adding/Removing Nodes/Edges

The most trivial way to add a (new) node to a graph is to simply include the node without adding an edge. Typically, a node is added along with connecting edges to the existing nodes. To remove a node, also remove its incident edges; this may cause the graph to become disconnected.

Adding a new edge between existing nodes or removing an existing edge modifies the set of edges without affecting the set of nodes.

Figure 6.3(a) shows the graph of Figure 6.1(a) with node a and its incident edges removed, and Figure 6.3(b) by adding edge $d \rightarrow k$ to Figure 6.3(a).

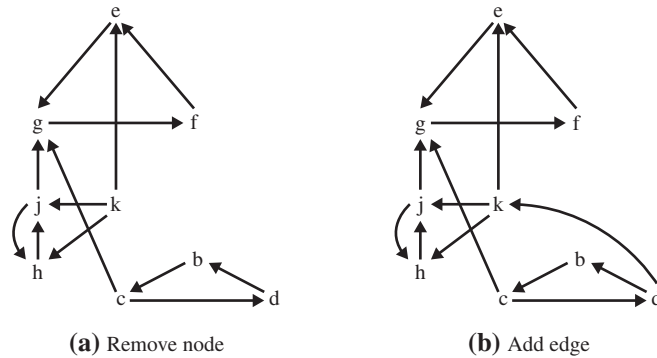


Figure 6.3 Removing node, adding edge to Figure 6.1.

6.2.6.2 Merging Nodes

It is often useful to treat a set of nodes X alike, by merging them into a single node x . The graph is then altered as follows: (1) modify the set of nodes by removing the nodes in X and adding a single node x , (2) remove every edge $u \rightarrow v$ where $u \in X$, $v \in X$, (3) replace every edge $u \rightarrow v$ where $u \notin X$, $v \in X$ by the edge $u \rightarrow x$, and (4) replace every edge $u \rightarrow v$ where $u \in X$, $v \notin X$ by the edge $x \rightarrow v$. Observe that this may create multiple edges between pairs of nodes if, for instance, node u , which is outside X , has edges to multiple nodes in X . Merging in an undirected graph is similarly defined.

A common application is in merging a strongly connected component X of graph G to a single node x to obtain graph G' ; see “Condensed graph” in Section 6.7.3.1 (page 313). Reachability among nodes is preserved in G' : (1) nodes within the strongly connected component X , being identified as a single node x in G' , are reachable from each other, (2) given $u \rightsquigarrow v$ in G where neither node is in X , $u \rightsquigarrow v$ in G' , (3) given $u \rightsquigarrow v$ in G where $v \in X$, $u \rightsquigarrow x$ in G' , and (4) given $u \rightsquigarrow v$ in G where $u \in X$, $x \rightsquigarrow v$ in G' . In Figure 6.1(a), $\{e, f, g\}$ is a strongly connected component. Figure 6.4 shows the effect of merging those nodes.

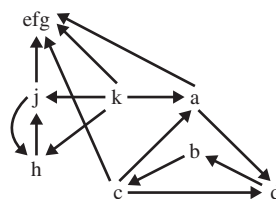


Figure 6.4 Merging nodes e, f and g of Figure 6.1(a).

6.2.6.3 Join or Replace Paths

The simplest operation on paths is to join two paths that share a common end point. That is, given $u \xrightarrow{p} v$ and $v \xrightarrow{q} w$, construct $u \xrightarrow{p} v \xrightarrow{q} w$, that is, $u \xrightarrow{pq} w$. As a special case, joining two paths, $u \xrightarrow{p} v$ and $v \xrightarrow{q} u$, results in a cycle $u \xrightarrow{pq} u$.

Another simple operation is to replace a portion of a path by a different path. Given $u \rightsquigarrow v \xrightarrow{p} w \rightsquigarrow x$ replace $v \xrightarrow{p} w$ by $v \xrightarrow{q} w$ to form $u \rightsquigarrow v \xrightarrow{q} w \rightsquigarrow x$. This is akin to taking a detour when part of a road network is closed.

Prune a path Any repeated occurrence of a node on a path can be eliminated, resulting in a simple path; Exercise 2 (page 364) asks for a proof of this fact. To see an example of this, suppose node w is repeated on path p between u and v , that is, $u \xrightarrow{p} w \xrightarrow{q} w \xrightarrow{r} v$. Then cycle q can be removed to get the path $u \xrightarrow{p} w \xrightarrow{r} v$ from u to v that has fewer occurrences of w . This step can be applied over and over to eliminate repeated occurrences of all nodes, resulting in a simple path. Clearly, the same argument applies if the path between u and v is a cycle.

6.2.6.4 Join Disjoint Cycles

Suppose there are two cycles that share no node. They can be joined to form a single cycle that includes all the nodes as follows: replace one edge in each cycle by two edges joining nodes of the two cycles. Specifically, consider two cycles $u \rightarrow v \xrightarrow{p} u$ and $x \xrightarrow{q} y \rightarrow x$. Replace edges $u \rightarrow v$ and $y \rightarrow x$ by $u \rightarrow x$ and $y \rightarrow v$ (see Figure 6.5(b)). The resulting cycle $u \rightarrow x \xrightarrow{q} y \rightarrow v \xrightarrow{p} u$ includes the nodes of both cycles. Further, if the original cycles are simple, so is the resulting cycle.

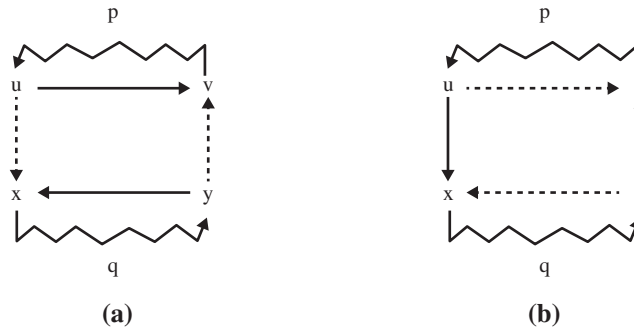


Figure 6.5 Join disjoint cycles.

6.2.6.5 Join Non-disjoint Cycles

Given two cycles that share node u , as in $u \xrightarrow{p} u$ and $u \xrightarrow{q} u$, form a single cycle that retains all the nodes and edges. This is simply $u \xrightarrow{p} u \xrightarrow{q} u$.

6.3 Specific Graph Structures

I study a few special kinds of graphs that often arise in practice. Such graphs have specific properties that may be exploited.

6.3.1 Tree

6.3.1.1 Rooted Tree

A connected directed graph is a tree, called a *rooted tree*, if it has a specific node called the *root* from which there is a unique path to every node. A rooted tree has no (directed) cycle because then there is more than one path to every node in the cycle, by repeating the nodes on the cycle an arbitrary number of times. I gave a short description of rooted trees in Section 2.1.2.7 (page 9). We have seen binary trees in the previous chapters. I show a rooted tree in Figure 6.6(b) where the root is *a* and the edges from parents to children are directed downward.

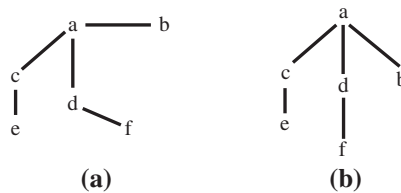


Figure 6.6 A free tree and the corresponding rooted tree.

There are many varieties of trees depending on the out-degrees of nodes and order among the children of a node (oldest to youngest, in conventional terms). Of particular interest in this book is *binary tree*, in which the out-degree of any node is at most two.

There are many important properties of trees. Most notable are: (1) there is exactly one path from a node to every node in its subtree, (2) any two subtrees are either disjoint (i.e., they have no common node or one is fully contained in the other), (3) the number of edges in a tree is one less than the number of nodes, and (4) in a tree in which there are l leaf nodes, nl non-leaf nodes and every non-leaf has exactly d children, $l = 1 + (d - 1) \times nl$.

Properties (1) and (2) follow directly from the definitions. Property (3) is seen easily as follows. Every node except the root can be mapped to a unique edge, the one from its parent to it. Conversely, every edge maps to a unique node, the one to which it is directed. Therefore, the number of edges equals the number of nodes minus the root. Proof of property (4) is by induction on nl , the number of non-leaf nodes. For $nl = 0$, $l = 1$. In order to increase nl , convert a leaf to a non-leaf with d children. This increases nl by 1 and l by $d - 1$.

6.3.1.2 Free Tree

A connected undirected graph without a cycle is a free tree. There is no designated root. A free tree is shown pictorially in Figure 6.6(a).

Designating a specific node as the root in a free tree creates a rooted tree, as follows. Direct all incident edges away from the root. For any node that has an incoming edge, direct all other incident edges away from it. Repeat this step until there are no undirected edges. You may convince yourself that (1) every node except the root has exactly one incoming edge and (2) every edge is directed. Since there is no cycle in the graph, there is no cycle after directing the edges. So, this procedure indeed creates a directed tree with the given root. Figure 6.6(b) shows the rooted tree, rooted at a , corresponding to the free tree of Figure 6.6(a).

6.3.2 Acyclic Graph

An *acyclic* graph, directed or undirected, has no cycle. See Figure 6.7 for an example of a directed acyclic graph.

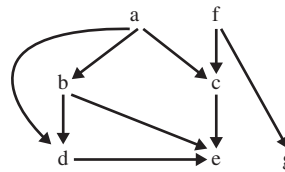


Figure 6.7 A directed acyclic graph.

A directed acyclic graph has at least one source and one sink node, that is, without incoming edges and outgoing edges, respectively. As seen in this figure, there may be multiple sources, a and f , and sinks, e and g .

A rooted tree is an acyclic directed graph where the root is the only source, every leaf node is a sink, and every node has a single predecessor. In a general acyclic directed graph, a node may have many predecessors, so, there may be multiple paths between any two nodes.

A connected undirected acyclic graph is a free tree.

Binary Decision Diagram Acyclic graphs arise in a number of applications. Of special interest in computer science is Binary Decision Diagram (BDD), which is used as a normal form for propositional boolean expressions. BDDs were introduced in Bryant [1992], and are now used extensively in computer-aided design (CAD) of circuits. We have seen conjunctive and disjunctive normal forms for boolean propositions in Section 2.5.4.1 (page 45). Such a normal form can be represented by a tree, as shown in Section 2.5.4.3 (page 49), Figure 2.10, and reproduced here in Figure 6.8.

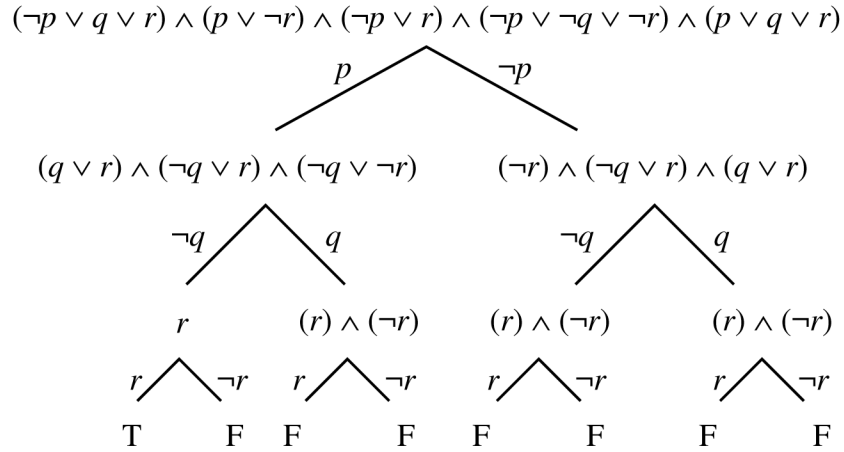


Figure 6.8 Successive reduction of a proposition by assigning values to its literals.

A tree representation is often wasteful because many subtrees tend to be identical. For example, all terminal nodes representing *true* may be combined, and similarly for *false*. Larger subtrees that are identical, such as for $r \wedge \neg r$, can also be combined. Further, any node whose two children are identical can be eliminated and replaced by one child. Applying these rules to the tree in Figure 6.8 yields the acyclic graph in Figure 6.9. Observe that the left and right subtrees for $(\neg r) \wedge (\neg q \vee r) \wedge (q \vee r)$ are identical; so, the entire subtree is replaced by the terminal node *F*.

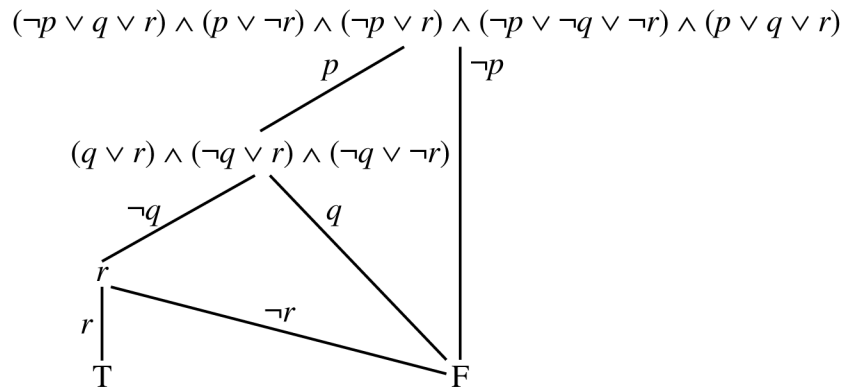


Figure 6.9 Compressing the tree in Figure 6.8 to an acyclic graph.

A BDD representation is not unique. A specific order of examining the variables results a unique BDD, p , q and then r , for the given example. Most operations on a boolean expression can be performed directly on its BDD representation.

6.3.3 Bipartite Graph

A graph is *bipartite* if it is possible to partition its nodes into two subsets, call them *black* and *white*, such that every edge is incident on a black node and a white node. That is, adjacent nodes have different colors. A bipartite graph may be undirected or directed. See Figure 6.10 for an example of an undirected bipartite graph; the nodes labeled with letters are black, and nodes with numbers are white.

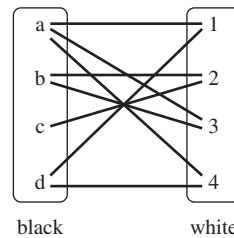


Figure 6.10 A bipartite graph.

A bipartite graph models relationships among objects of two different types. For example, a factory may have several machines and several jobs to perform. Not every machine can perform every job. Represent a machine by a black node and a job by a white node, and draw an edge between a black and white node only if the corresponding machine can perform the given job. Such a representation is useful in scheduling jobs on machines efficiently.

A directed graph is bipartite iff its underlying undirected graph is bipartite. This is because the adjacency relationship ignores edge directions. The following theorem describes a fundamental property of bipartite graphs.

Theorem 6.1 An undirected graph is bipartite iff all its cycles have even lengths.

Proof. First, I show that any cycle in a bipartite graph has even length. Color the nodes black and white so that adjacent nodes have different colors. Then in any cycle each edge is incident on a black and a white node, so the node colors alternate along the cycle. Hence, the cycle length is even.

Next, I show that an undirected graph in which all cycles are of even length is bipartite. I assume that the graph is connected, otherwise prove the result for each connected component separately. Choose an arbitrary node as *root*. I claim that for any node x all paths between *root* and x are of even length or all paths are of odd

length. Otherwise, if $root \stackrel{p}{\sim} x$ is of even length and $root \stackrel{q}{\sim} x$ is of odd length, then the cycle $root \stackrel{p}{\sim} x \stackrel{q}{\sim} root$ is of odd length, violating the given condition.

Now, color x white if all its paths from $root$ are of even length and black otherwise. So, $root$ is white because it is on a path of length 0 from itself. For any edge $x-y$, I claim that x and y have different colors, so the graph is bipartite. If x is white, so at even length from $root$, the path $root \stackrel{p}{\sim} x-y$ is of odd length, so y is black. ■

Observe that an undirected graph without a cycle, that is, a tree, meets the condition of this theorem, so it is bipartite.

6.3.4 Euler Path and Cycle

Graph theory originated with a puzzle known as the Seven Bridges of Königsberg.¹ It was solved by the great mathematician Leonhard Euler in 1735. Königsberg was a Prussian city in Euler's time. It had seven bridges over the Pregel River, which forks and rejoins as shown in schematic in Figure 6.11. The landmasses are numbered from 1 to 4, and the seven bridges are shown as rectangular boxes joining the landmasses.

The puzzle was to determine if it is possible to walk a continuous route so that each bridge is crossed exactly once. Euler translated the problem to a form that is independent of the shapes of the landmasses, retaining only the connection information between them, a form we recognize today as a graph. He proved a necessary condition for the existence of such a walk, and consequently that there is no such walk for the bridges of Königsberg. Later mathematicians proved that Euler's condition is also sufficient.

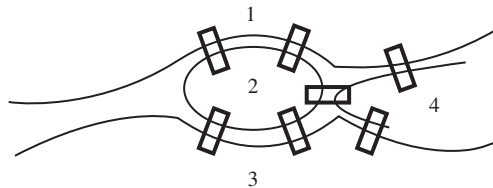


Figure 6.11 Schematic of the seven bridges of Königsberg.

Figure 6.12 converts the problem to one in graph theory. Each landmass is a node and each bridge an undirected edge that joins two nodes. The problem then becomes: does there exist a path in the given undirected graph that includes every edge exactly once? A related problem is: is there a cycle that includes every edge exactly once? Such a path and cycle, if they exist, are known, respectively, as *Euler path* and *Euler cycle*.

1. This is why puzzles are important!

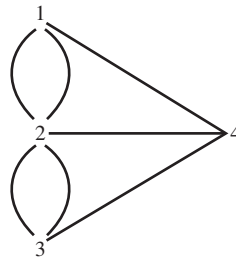


Figure 6.12 Graph representation of the bridges of Königsberg.

First, consider the Euler cycle problem. Assume that the graph is connected; otherwise, there is no solution. The following observation is essential in establishing the main result, Theorem 6.2.

Observation 6.1 Consider a path (or cycle) in an undirected graph in which no edge is repeated though nodes may be repeated. Then every interior node has an even number of incident edges in the path, and every exterior node has an odd number of incident edges in the path.

Proof. Every occurrence of an interior node on a path has two distinct incident edges on it, with the nodes preceding and succeeding it in the path. Therefore, for any number of occurrences of a node in the path, an even number of edges are incident on it in the path.

An exterior node has either no preceding or no succeeding node, so, according to the preceding argument, it has an odd number of edges incident on it. ■

Theorem 6.2 An Euler cycle exists in an undirected graph iff every node has even degree.

Proof. First, I show that if there is an Euler cycle, then every node has even degree. The Euler cycle includes all edges, hence all nodes of the graph. Further, every node is an interior node of a cycle. From Observation 6.1, every node has an even number of incident edges in the cycle. Since the cycle includes all the graph edges, every node has even degree. This part of the theorem was proved by Euler.

Next, I show the converse, that if each node has even degree, then there is an Euler cycle. The proof is by induction on the number of edges. If the graph has no edge, since the graph is connected, it has a single node and the (vacuous) cycle that includes this node includes all the edges.

For the inductive step, consider a graph with non-zero number of edges. First, I show that it includes a cycle. Clearly, it has a non-empty path because there are edges. If the path is not a cycle, it has exterior nodes each with odd number of incident edges on the path. So, some incident edge on an exterior node is not on

the path because every node has even degree. Using this edge, the path can be extended. Applying this step repeatedly, continue extending the path. Since it cannot be extended indefinitely, there will be no exterior node eventually, that is, the path is a cycle.

If the constructed cycle includes all the edges, then the result is proved. Otherwise, the edges outside the cycle constitute a graph (over the given set of nodes) where each node has even degree. This graph may have multiple components where each component is smaller than the original graph. Inductively, each component has an Euler cycle.

The final step is to join the cycles into a single cycle. Start with any cycle. Since the original graph is connected, it shares a node with some other cycle. Join the two cycles as given in Section 6.2.6.5 (page 273) under “Join non-disjoint cycles”. Repeat until all cycles are joined. ■

The general result for Euler paths is given in the following theorem; its proof is similar to that of Theorem 6.2.

Theorem 6.3 An Euler path with distinct exterior nodes exists in an undirected graph iff there are exactly two nodes with odd degree. ■

The problem asked for the Königsberg bridges in Figure 6.12 is if there exists an Euler path. Every interior node on an Euler path has even degree, but no node in the given graph has even degree; so, there is no solution to the Königsberg bridge problem.

Euler path/cycle in directed graph As in undirected graphs, a cycle that includes all the edges exactly once in a directed graph is an Euler cycle, and analogously for Euler path.

Theorem 6.4 There is an Euler cycle in a strongly connected directed graph iff every node has the same in-degree and out-degree. There is an Euler path in a strongly connected directed graph iff there is exactly one node whose in-degree is one less than its out-degree, one node whose in-degree is one more than its out-degree and every other node has the same in-degree and out-degree.

6.3.5 Hamiltonian Path and Cycle

A *Hamiltonian* path in a graph, directed or undirected, is a path that includes each node exactly once. If the exterior nodes of the path are identical, then the path is a cycle, called a *Hamiltonian cycle*. Hamiltonian path/cycle is simple because there is no repeated node along the path/cycle. Figure 6.12 (page 279) has a Hamiltonian path $a-b-d-c$ and a Hamiltonian cycle $b-d-c-a-b$.

There is no simple condition, analogous to Euler path, for the existence of a Hamiltonian path in a graph. In fact, it is an NP-complete problem.

6.4 Combinatorial Applications of Graph Theory

I consider a few problems that have no apparent connection to graph theory. But viewed as graph problems they readily yield simple solutions.

6.4.1 A Puzzle About Coloring Grid Points

The following problem (given verbatim) is from [Dijkstra \[1996b\]](#). A “grid line”, below, is either a horizontal or vertical line as in a graph paper, and a “grid point”, denoted by $\bullet$, is the point of intersection of a horizontal and a vertical grid line (see Figure 6.13(a)).

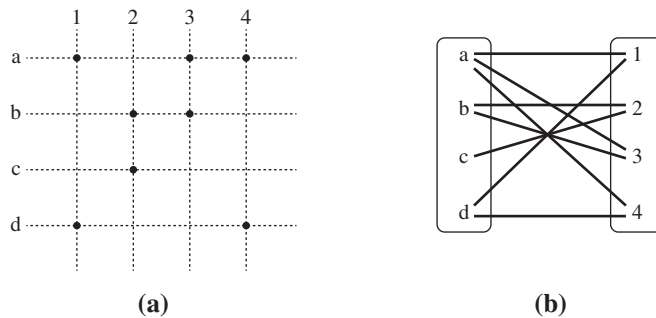


Figure 6.13 Points on grid lines and the corresponding bipartite graph.

Show that, for any finite set of grid points in the plane, we can colour each of the points either red or blue such that on each grid line the number of red points differs by at most 1 from the number of blue points.

Dijkstra gives two different solutions for this problem, introducing terms like “snake” and “rabbit”. He motivates the steps in the development of each solution. He does not appeal to graph theory, which would have reduced it to a simpler problem. I encourage the reader to tackle this problem before looking at the solution.

Solution Construct a graph in which each grid line is a node and each grid point an edge connecting the two nodes corresponding to the grid lines to which it belongs (see Figure 6.13(b)). It is required to color each edge red or blue so that the number of red and blue edges incident on any node differ by at most one.

First, observe that a node corresponds to either a vertical or a horizontal grid line and each edge connects a “vertical” node to a “horizontal” node. So, the graph

is bipartite. Each cycle in a bipartite graph has an even number of edges (see Theorem 6.1, page 277). So, the edges of a cycle may be colored alternately, red and blue, and such a coloring does not affect the outcome at any node (see Figure 6.14(a) where R and B denote red and blue). Hence, we may remove the cycles from the graph (in arbitrary order) and solve the coloring problem over the remaining edges (see Figure 6.14(b)).

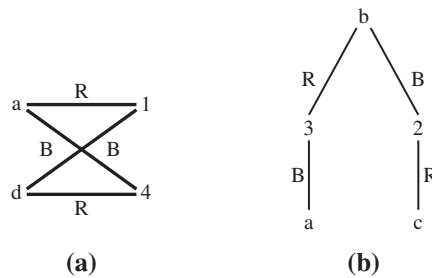


Figure 6.14 Coloring the edges of a bipartite graph.

After removing the cycles, we are left with an acyclic undirected graph, that is, a *tree* (or, possibly, a *forest*). The different trees in a forest may be independently colored, as follows. Construct a rooted tree by choosing an arbitrary node as the root; in Figure 6.14(b) we have chosen b to be the root. Order the children of every node arbitrarily. Color the edges incident on the root alternately, red and blue, using the order of its children. For any other node whose edge to its parent is colored red/blue, color the edges to its children alternately starting with blue/red.

6.4.2 Domino Tiling

Problem description The following problem is similar to the trimino tiling problem of Section 3.2.2.1 (page 90), except dominos are used instead of triminos for tiling. A domino covers two *adjacent* cells in a board, either horizontally or vertically.

Given is an 8×8 board in which two cells are *closed* and the other cells are *open*. It is required to place the dominos on the board so that there is exactly one domino over each open cell and none over a closed cell.

There is an elegant proof that if two corner cells are closed then there is no solution. To see this, imagine that the board is a chess board so each cell is either black or white and adjacent cells have different colors. Then each domino covers one black and one white cell; therefore, any tiling of the board covers equal number of black and white cells. The two corner cells have the same color; so, it is impossible to tile the board if they are both closed. The same argument can be used to prove

that if two cells of the same color are closed anywhere in the board then there is no possible tiling.

I prove a converse result, that if any two cells of different colors are closed, then it is possible to tile the board. Actually, I prove the result for any board with even number of rows and at least two columns. The restrictions on the number of rows and columns is necessary. Since the board has to have an equal number of black and white cells, the total number of cells is even; so the number of rows or columns is even, and, without loss in generality, I have chosen the number of rows to be even. There should be at least two columns because in a single column if two adjacent cells are closed so that both sides around the closed cells have an odd number of open cells, then there is no possible tiling. Henceforth, call a board with even number of rows and more than one column a *proper board*.

Convert the problem to one over an undirected graph: each cell is a node and adjacent cells have an edge between them. A domino is placed on an edge. Then it covers the cells corresponding to both of its incident nodes. A tiling C is a set of edges so that (1) each open cell is incident on exactly one edge in C and (2) no edge in C is incident on a closed cell.

Hamiltonian cycle over a proper board I start by proving a seemingly unrelated fact: a proper board has a Hamiltonian cycle. I actually prove a slightly stronger result, that a proper board has a Hamiltonian cycle in which all nodes of the top row are connected to their adjacent nodes in that row. The proof is by induction on the number of rows.

In the base case, there are two rows. Construct a simple cycle that connects all the cells, where each cell is denoted by $\bullet$ and a domino by an edge, as shown in Figure 6.15. Observe that all nodes of both top and bottom row are connected to their adjacent nodes in that row.

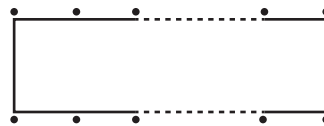


Figure 6.15 Hamiltonian cycle in a proper board of two rows.

For the general case, construct Hamiltonian cycles for the first two rows as given in Figure 6.15, and, inductively, for the remaining rows (see Figure 6.16(a)). The bottom Hamiltonian cycle for the inductive case has the two leftmost cells in its top row joined by an edge, from the induction hypothesis. Join the two disjoint cycles in Figure 6.16(b) by removing two edges, shown by fine dotted lines, and joining

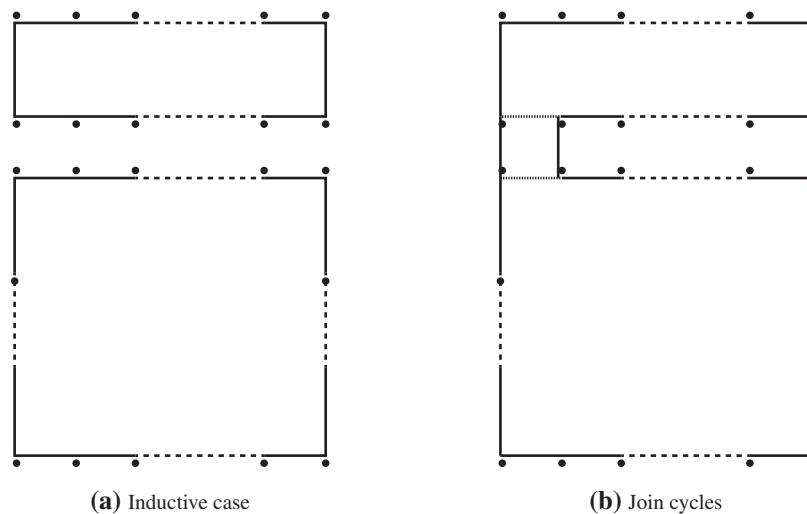


Figure 6.16 Hamiltonian cycle for the general case.

their endpoints, shown by solid lines (see joining disjoint cycles in Section 6.2.6.4, page 273).

Tiling a proper board with closed cells The proof of tiling is now simple. Denote the two closed cells by B and W (for black and white). The Hamiltonian cycle in this graph is of the form $B \overset{p}{\rightsquigarrow} W \overset{q}{\rightsquigarrow} B$. Each of the paths, p and q , has an even number of nodes since they have nodes of different color as their exterior nodes. On each path remove the edges alternately starting from any one end. The remaining edges form the required tiling: every open node has exactly one incident edge and neither B nor W has an incident edge. Figure 6.17 shows a Hamiltonian cycle in (a), where the lowercase b and w denote black and white cells and B and W the closed cells, and the tiling itself in (b) of that figure.

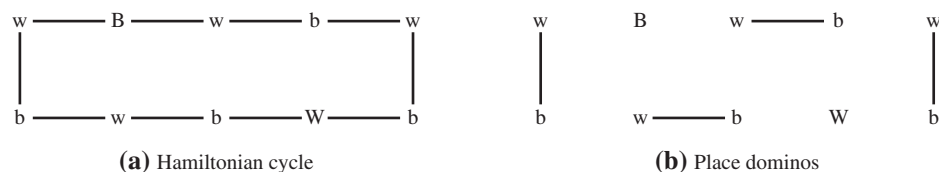


Figure 6.17 Domino placement on a Hamiltonian path excluding B and W .

6.4.3 Fermat's Little Theorem

The following theorem, known as Fermat's little theorem, is important in number theory. The typical proofs of this result use elementary number-theoretic results.

The proof below, using elementary graph theory, is a rewriting of a short proof that appears in [Dijkstra \[1980\]](#). Actually, the same proof can be written without appealing to graph theory [see [Misra 2021](#)].

Theorem 6.5 For any natural number n and prime number p , $n^p - n$ is a multiple of p .

Proof. For $n = 0$, $n^p - n$ is 0, hence a multiple of p . For positive integer n , take an alphabet of n symbols and construct a graph as follows: (1) each node of the graph is identified with a distinct string of p symbols, called a “word”, and (2) there is an edge $x \rightarrow y$ if rotating word x by one place to the left yields y . Observe:

1. No node is on two simple cycles because every node has a single successor and a single predecessor (which could be itself).
2. Each node is on a cycle of length p because successive p rotations of a word transforms it to itself.
3. Every simple cycle's length is a divisor of p , from (2). Since p is prime, the simple cycles are of length 1 or p .
4. A cycle of length 1 corresponds to a word of identical symbols. So, exactly n distinct nodes occur in cycles of length 1. The remaining $n^p - n$ nodes occur in simple cycles of length p .
5. A simple cycle of length p , from the definition of a simple cycle, has p distinct nodes. From (4), $n^p - n$ is a multiple of p . ■

6.4.4 De Bruijn Sequence

What is the shortest binary sequence that cyclically contains every binary sequence of length n as a substring, for a positive integer n ? For $n = 1$, there are two such sequences, 01 and 10. For $n = 2$, there are four binary sequences of length 2, so an 8-bit string suffices. But we can do better, the 4-bit sequence 0011 cyclically contains every 2-bit sequence *exactly* once, in order from left: $\langle 00, 01, 11, 10 \rangle$.

The general problem asks the same question for sequences of length n over any finite alphabet of size k , not just over the binary alphabet. There are k^n “words” (sequences) of length n over this alphabet. And there exist $\frac{(k!)^{k^n-1}}{k^n}$ sequences, each of length k^{n+1} , that include all the words. Such sequences are known as *de Bruijn sequence* in honor of N. G. de Bruijn who published a significant paper on the problem in 1946. The problem had been studied earlier for special cases; the earliest mention in antiquity is in the works of the Sanskrit grammarian Panini for $k, n = 2, 3$.

There are many applications of de Bruijn sequence. One application is to test a finite state machine by supplying all possible words to it as stimuli. The shortest string that supplies such stimuli is a de Bruijn sequence.

Construction of de Bruijn sequence There is an elegant technique to construct de Bruijn sequence of any length. For words of length 1, that is, $n = 1$, any sequence that contains all symbols exactly once is a de Bruijn sequence. For $n + 1$, where $n \geq 1$, (1) construct a graph of k^{n+1} edges as described below, where each edge label is a symbol of the alphabet, and (2) construct an Euler cycle from the graph. The sequence of labels on the edges of the Euler cycle, of length k^{n+1} , is a de Bruijn sequence.

Construction of the graph, called a *de Bruijn graph*, for any k and n , $k \geq 1$, $n \geq 1$, is as follows. The graph has k^n nodes and each node is labeled with a word of length n . From node labeled ax , whose first symbol is a , draw an edge labeled b to node xb whose last symbol is b : $ax \xrightarrow{b} xb$. See Figure 6.18(a), (b) and (c) for $n = 1$, $n = 2$ and $n = 3$, respectively, over the binary alphabet.

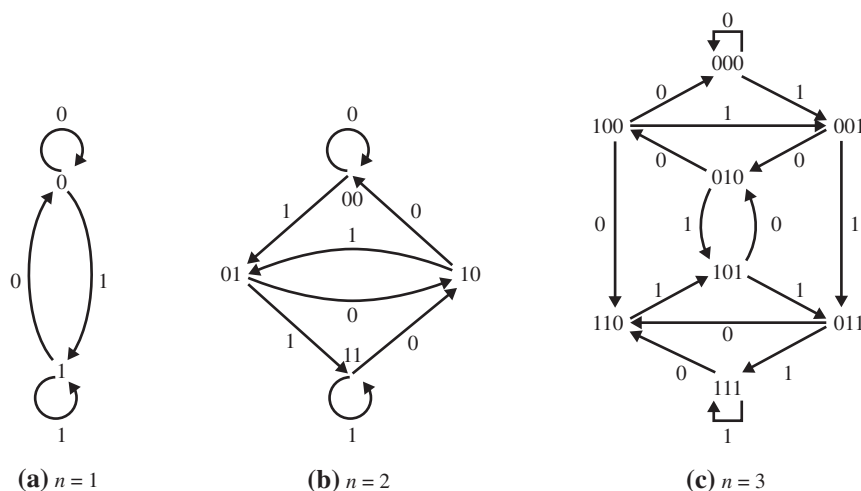


Figure 6.18 de Bruijn graphs for $n = 1, 2, 3$ over the binary alphabet.

Henceforth, each word and path is of length n unless otherwise specified. A path label is the sequence of edge labels along it.

We need one property of these graphs: for any node x and word y , there is a simple path $x \xrightarrow{y} y$ in the graph. For example, in Figure 6.18(b) from any node there is a path ending at node 10 that is labeled 10, and in Figure 6.18(c) from any node there is a path ending at node 101 that is labeled 101.

Observation 6.2 For any two nodes x and y , $x \xrightarrow{y} y$.

Proof. This should be evident from the construction. For any edge, a symbol is dropped from the prefix of the starting node and the edge label added as a suffix to

the ending node. Further, every node has outgoing edges that are labeled with all the symbols of the alphabet. For node with label $\langle x_1 x_2 \cdots x_n \rangle$ and word $\langle y_1 y_2 \cdots y_n \rangle$, we have:

$$\langle x_1 x_2 \cdots x_n \rangle \xrightarrow{y_1} \langle x_2 \cdots x_n y_1 \rangle \xrightarrow{y_2} \cdots \langle x_n y_1 y_2 \cdots y_{n-1} \rangle \xrightarrow{y_n} \langle y_1 y_2 \cdots y_n \rangle.$$

So, $\langle x_1 x_2 \cdots x_n \rangle \xrightarrow{\langle y_1 y_2 \cdots y_n \rangle} \langle y_1 y_2 \cdots y_n \rangle.$ ■

Theorem 6.6 The de Bruijn graph for any k and n has an Euler cycle of length k^{n+1} , which is a de Bruijn sequence.

Proof. Observe for the constructed de Bruijn graph:

1. There is a path from any node x to any other node y , from Observation 6.2. So the graph is strongly connected.
2. Each node has both in-degree and out-degree of k , from the construction of the graph.
3. From (1,2) and Theorem 6.4 (page 280), there is an Euler cycle. The graph has k^n nodes each with out-degree k ; so, the Euler cycle has k^{n+1} edges.
4. I show that the sequence of edge labels along the Euler cycle is a de Bruijn sequence that contains every $(n + 1)$ -long word, xb , as a substring. Consider the outgoing edge labeled b from node x . This edge appears somewhere along the Euler cycle. The path of length n preceding it along the cycle has label x , since it ends at node x , according to Observation 6.2. So, string xb appears somewhere within the Euler cycle. ■

As an example, apply the construction to Figure 6.18(a). An Euler cycle is: **0 0 0 1 1 1 1 0 0**, where the node labels are in boldface and the edge labels in normal face. The edge labels along the cycle is the sequence $\langle 0, 1, 1, 0 \rangle$, a de Bruijn sequence of length 4 that includes every binary string of length 2 cyclically.

6.4.5 Hall's Marriage Theorem

Hall's [1935] marriage theorem is applicable in several combinatorial problems. Given is a finite bag B whose members are sets T_i , $0 \leq i < n$. A system of distinct representatives (SDR) is a set of distinct elements t_i , $0 \leq i < n$, where $t_i \in T_i$. Note that B is a *bag* of sets instead of a *set* of sets because some of the sets in B may be identical. Hall gives the necessary and sufficient condition for the existence of an SDR in bag B : for every subbag of B , the number of elements in its constituent sets is at least the size (the number of members) of the subbag.

I prove this theorem using graph theory. Represent B by a bipartite graph with node sets X and Y : each element of B , a set, is a node in X and an element in any set of B is a node in Y . Edge (x, y) , where $x \in X$ and $y \in Y$, represents that element y belongs to the set in B represented by x .

A *matching* is a set of edges that have no common incident nodes. A (X, Y) matching is a matching in which every node of X is incident on some edge in the matching. A (X, Y) matching represents an SDR. Hall's theorem gives the necessary and sufficient condition for the existence of a (X, Y) matching.

Hall condition (HC) Subset S of X meets HC if the number of neighbors of S is greater than or equal to the size of S .

Theorem 6.7 There is a (X, Y) matching if and only if every subset of X meets HC.

Proof. The proof in one direction, that if there is a (X, Y) matching every subset of X meets HC, is straightforward. I prove the converse of the statement by induction on the size of set X . If X is empty, there is a trivial matching. For the general case, assume, using induction, that there is a matching over all nodes of X except one node r .

Henceforth, $u \overset{n}{-} v$ and $u \overset{m}{-} v$ denote, respectively, that (u, v) is a non-matching edge and (u, v) a matching edge. An *alternating path* is a simple path of alternating matching and non-matching edges. Let Z be the subset of nodes of X that are connected to r by an alternating path and Z' the neighbors of Z , a subset of Y .

Every node of Z except r is connected to a unique node in Z' by a matching edge, from the induction hypothesis; so, there are $|Z| - 1$ nodes in Z' that are so connected. Since Z meets HC, $|Z'| \geq |Z|$. Therefore, there is a node v in Z' that is not connected to any node in Z by a matching edge. I show that v is not incident on any matching edge.

Let v be a neighbor of u in Z , so $u \overset{n}{-} v$. Since $u \in Z$, there is an alternating path between r and u ; extend the path to include edge $u \overset{n}{-} v$, as shown below in P . I color the matching edges blue and non-matching edges red for emphasis.

$$P : r = x_0 \overset{n}{-} y_0 \overset{m}{-} x_1 \cdots x_i \overset{n}{-} y_i \overset{m}{-} x_{i+1} \cdots x_t \overset{n}{-} y_t \overset{m}{-} x_{t+1} = u \overset{n}{-} v.$$

If v is incident on a matching edge, say $v \overset{m}{-} w$, then $w \notin Z$ because v is not connected to any node in Z by a matching edge. However, we can extend P as follows which shows that r is connected to w by an alternating path, so $w \in Z$, a contradiction.

$$P' : r = x_0 \overset{n}{-} y_0 \overset{m}{-} x_1 \cdots x_i \overset{n}{-} y_i \overset{m}{-} x_{i+1} \cdots x_t \overset{n}{-} y_t \overset{m}{-} x_{t+1} = u \overset{n}{-} v \overset{m}{-} w.$$

So, we conclude that v is not incident on any matching edge.

Flip the edge labels in P from n to m and m to n to obtain a matching that includes all previously matched nodes of X and now includes r , so all nodes of X are in the matching.

$$r = x_0 \xrightarrow{m} y_0 \xrightarrow{n} x_1 \cdots x_i \xrightarrow{m} y_i \xrightarrow{n} x_{i+1} \cdots x_t \xrightarrow{m} y_t \xrightarrow{n} x_{t+1} = u \xrightarrow{m} v.$$

This completes the inductive proof. ■

6.5 Reachability in Graphs

Reachability is a fundamental problem in graph theory. Given is a finite directed graph with a designated *root* node. It is required to identify all nodes that are reachable via a directed path from *root*. There is an analogous problem for undirected graphs whose solution is very similar. For the graph shown in Figure 6.19(a), the set of reachable nodes for each node appears in Figure 6.19(b).

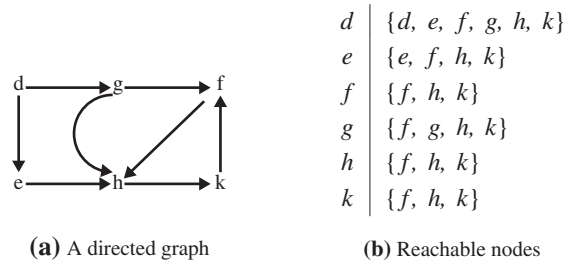


Figure 6.19 A directed graph and its lists of reachable nodes.

There are many well-known algorithms to solve the reachability problem, such as breadth-first and depth-first traversal, which are discussed in Section 6.6. In this section, I propose and prove an abstract algorithm that includes the known algorithms as special cases. The proposed algorithm is non-deterministic. A non-deterministic algorithm embodies a family of deterministic algorithms. Proving the correctness of a non-deterministic algorithm proves the correctness of the entire family. The correctness of the abstract reachability algorithm turns out to be somewhat involved; so, I prove it in detail.

6.5.1 Definition and Properties of Reachability Relation

Definitions Let x be a node and S a set of nodes in a given graph. Write $\text{succ}(\{x\})$ for the set of successor nodes of x . Below, I formally define $R(\{x\})$ as the set of nodes reachable from x and $R_i(\{x\})$ as the set of nodes reachable from x via a path (not necessarily a simple path) of length i , $i \geq 0$. Observe that $R_{i+1}(\{x\})$ is the set of nodes that are successors of the nodes in $R_i(\{x\})$.

Extend the notations for succ , R_i and R so that each function has a *set* of nodes as argument and the function value is the union of the results for individual nodes in the argument. That is, for a set of nodes S , $R(S) = (\cup x : x \in S : R(\{x\}))$. Therefore, for sets of nodes S and T , $R(S \cup T) = R(S) \cup R(T)$. Also, given $S \subseteq T$, $R(S) \subseteq R(T)$.

Formal definitions of R_i and R , $i \geq 0$:

1. $R_i(\{\}) = \{\}$.
2. $R_0(\{x\}) = \{x\}$.
3. $R_{i+1}(\{x\}) = \text{succ}(R_i(\{x\}))$.
4. $R(\{x\}) = (\cup n : n \geq 0 : R_n(\{x\}))$.

Properties of R

- P1. $R(\{\}) = \{\}$.
- P2. $R(S) = S \cup \text{succ}(R(S)) = S \cup R(\text{succ}(S))$.
- P3. $S \subseteq R(S)$ and $\text{succ}(R(S)) \subseteq R(S)$, from (P2).
- P4. $R(R(S)) = R(S)$.
- P5. $(\text{succ}(S) \subseteq S) \equiv (S = R(S))$.

Properties P1 through P4 are proved easily from the definition. I prove property P5. Proof is by mutual implication.

- $(S = R(S)) \Rightarrow (\text{succ}(S) \subseteq S)$:

$$\begin{aligned}
 & \text{succ}(S) \\
 = & \quad \{\text{From the antecedent, } S = R(S). \text{ Replace } S \text{ by } R(S).\} \\
 & \text{succ}(R(S)) \\
 \subseteq & \quad \{\text{from P2, } \text{succ}(R(S)) \subseteq R(S)\} \\
 & R(S) \\
 = & \quad \{\text{antecedent}\} \\
 & S
 \end{aligned}$$

- $(\text{succ}(S) \subseteq S) \Rightarrow (S = R(S))$:

From P3, $S \subseteq R(S)$. So, it suffices to prove $(\text{succ}(S) \subseteq S) \Rightarrow (R(S) \subseteq S)$.

I prove from $(\text{succ}(S) \subseteq S)$ that $R_n(S) \subseteq S$ for all n , $n \geq 0$.

Therefore, $R(S) = (\cup n : n \geq 0 : R_n(S)) \subseteq S$.

The proof of $R_n(S) \subseteq S$ for all n is by induction on n . For $n = 0$, $R_0(S) = S$. Assume inductively that the result holds for some n , $n \geq 0$.

$$\begin{aligned}
 & R_{n+1}(S) \\
 = & \quad \{\text{definition}\} \\
 & \text{succ}(R_n(S)) \\
 \subseteq & \quad \{\text{Induction: } R_n(S) \subseteq S\}
 \end{aligned}$$

$$\begin{array}{c} \text{succ}(S) \\ \subseteq \quad \{\text{antecedent: } \text{succ}(S) \subseteq S\} \\ S \end{array}$$

■

Relation R is also known as the reflexive and transitive closure of the succ relation (see Section 2.3.5, page 31). Section 6.8.2 (page 319) includes an efficient algorithm for the computation of transitive closure.

6.5.2 An Abstract Program for Reachability

I use two sets of nodes, marked and vroot , in the abstract algorithm. If a node is marked , it is reachable from root . The nodes in vroot are also reachable from root , though they are not yet marked. Initially, all nodes are unmarked , that is, not in the set marked , and finally, only and all reachable nodes from root are marked.

Initially, marked is empty and vroot contains only root . At any step if vroot is non-empty, an arbitrary node u is chosen and removed from it, u is marked, and all its unmarked successors are added to vroot . These steps are repeated until vroot becomes empty.

Predicate $\text{marked} \cup R(\text{vroot}) = R(\text{root})$ is perpetually true, though it is not an invariant. Hence, at the termination of the algorithm, when vroot is empty, $\text{marked} = R(\text{root})$, that is, only and all reachable nodes from root are then marked.

Abstract program for reachability

```

marked, vroot := {}, {root};
while vroot ≠ {} do
  u :∈ vroot ;
  marked := marked ∪ {u};
  vroot := (vroot ∪ succ(u)) − marked
enddo

```

Figure 6.20

6.5.3 Correctness Proof

Partial correctness The invariant required for the proof of the program in Figure 6.20 is not obvious. Though $\text{marked} \cup R(\text{vroot}) = R(\text{root})$ is perpetually true, it is not an invariant. I prove the invariance of:

$$\text{Inv} :: \text{root} \cup \text{succ}(\text{marked}) \subseteq \text{marked} \cup \text{vroot} \subseteq R(\{\text{root}\}).$$

Proof of the verification conditions of this invariant is in Appendix 6.A (page 371). Here, I show that the program terminates and $\text{marked} = R(\{\text{root}\})$ holds at termination. Using Inv we can deduce at termination:

$$\begin{aligned}
& \text{Inv} \wedge \text{vroot} := \{\} \\
\Rightarrow & \{\text{expand } \text{Inv} \text{ and simplify}\} \\
& \text{root} \cup \text{succ}(\text{marked}) \subseteq \text{marked} \subseteq R(\{\text{root}\}) \\
\Rightarrow & \{\text{from } \{\text{root}\} \subseteq \text{marked}, R(\{\text{root}\}) \subseteq R(\text{marked})\} \\
& R(\{\text{root}\}) \subseteq R(\text{marked}), \text{succ}(\text{marked}) \subseteq \text{marked} \subseteq R(\{\text{root}\}) \\
\Rightarrow & \{\text{use Property P5 on } \text{succ}(\text{marked}) \subseteq \text{marked}\} \\
& R(\{\text{root}\}) \subseteq R(\text{marked}), R(\text{marked}) = \text{marked}, \text{marked} \subseteq R(\{\text{root}\}) \\
\Rightarrow & \{\text{simplify}\} \\
& R(\{\text{root}\}) \subseteq \text{marked} \subseteq R(\{\text{root}\}) \\
\Rightarrow & \{\text{simplify}\} \\
& \text{marked} = R(\{\text{root}\})
\end{aligned}$$

Termination The termination of the program is obvious. In each iteration, a new node is added to marked ; so the number of iterations is finite for a finite graph. A formal proof can be devised as follows: (1) prove the invariant that every node in vroot is unmarked, that is, $\text{vroot} \cap \text{marked} = \{\}$, (2) since the assignment $\text{marked} := \text{marked} \cup \{u\}$ is executed when $u \in \text{vroot}$, the size of marked strictly increases, or the number of unmarked nodes decreases in each iteration.

6.6 Graph Traversal Algorithms

A graph traversal algorithm *visits* and *makes* all graph nodes in some particular order; in the process it also has to visit every edge. The traversal can be used to process information stored at every node/edge of the graph. A traversal typically induces a tree structure over the nodes by choosing a subset of edges. The induced tree structure provides a basis for recursive algorithms and reasoning by induction; normally, a graph does not yield to recursive algorithms or reasoning by induction because of the existence of cycles.

We study two traversal algorithms in this section, *breadth-first* and *depth-first* traversal. I show traversal algorithms for connected undirected graph and for directed graphs in which all nodes are reachable from a designated root node. I generalize these algorithms in Section 6.6.2.5 (page 302) where the connectivity and reachability assumptions are dropped. I show one example, identifying the strongly connected components of a directed graph, in Section 6.7.3 (page 311), that is based on the tree structure induced by depth-first traversal; this tree structure was introduced in Tarjan [1972].

Convention I assume in this section that there is no edge, $u \rightarrow u$, from a node to itself (such an edge is called a *self loop*) in a directed graph. There is no technical difficulty in admitting such an edge, but it is easier to state and prove some of the results in the absence of self loops.

6.6.1 Breadth-first Traversal

I describe the breadth-first traversal algorithm, first for directed graphs and later for undirected graphs and trees.

Breadth-first traversal of directed graph Given is a directed graph with a specified *root* node. The *level* of node u , reachable from *root*, is the minimum number of edges on a path from *root* to u . Breadth-first traversal marks the reachable nodes by levels, all nodes of level 0 followed by nodes of level 1, and so on. Initially, all nodes are unmarked. Then *root*, the only node of level 0, is marked. Next, all unmarked nodes reachable from *root* are marked. In every iteration, the unmarked successors of level i nodes are identified as level $i + 1$ nodes and then marked. The execution terminates when no more nodes can be marked.

Consider the graph of Figure 6.1, which I repeat here for easy reference in Figure 6.21. Its breadth-first traversal is shown in Figure 6.22 using node k as the *root*. The nodes are visited level by level and from left to right within a level. A solid edge shows the visit to an unmarked node, which is then marked, and a dotted edge shows visiting a node that is already marked.

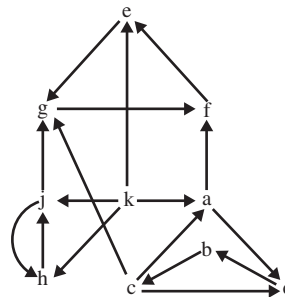


Figure 6.21 The directed graph from Figure 6.1.

As I have remarked in the introduction to Section 6.5, the breadth-first marking algorithm is a special case of the abstract algorithm for reachability in Figure 6.20 (page 291). I refine the abstract algorithm in Figure 6.23 in which *vroot* is implemented as a queue (a queue is a sequence, so its elements are within angled brackets, $\langle \rangle$). Initially, *vroot* contains only *root*. Iterations continue as long as the queue is non-empty. In every iteration, the head of the queue, u , is removed from the queue

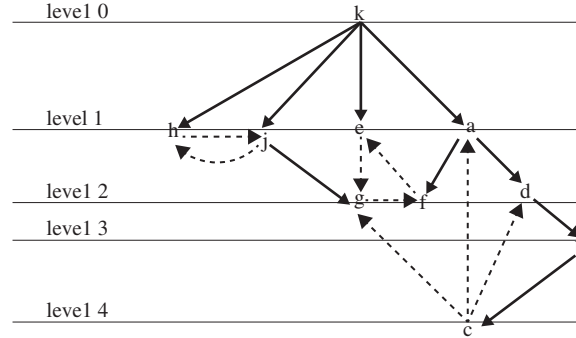


Figure 6.22 Breadth-first traversal of the graph from Figure 6.1.

Breadth-First traversal of directed graph

```

marked, vroot := {}, ⟨root⟩;
while vroot ≠ ⟨⟩ do
  u, vroot, marked := vroot.head, vroot.tail, marked ∪ {u} ;
  for v ∈ succ(u) − marked do
    parent[v] := u ;
    vroot := vroot ++ ⟨v⟩
  endfor
enddo

```

Figure 6.23

and *marked*. All unmarked successors of *u* are appended to the end of the queue; *vroot* := *vroot* ++ ⟨*v*⟩ appends node *v* to the end of *vroot*.

I augment this scheme to induce a tree structure explicitly. Any node *u* that is removed from *vroot* is the parent of its every unmarked successor.

It follows from the proof in Section 6.5.3 (page 291) that *marked* is the set of reachable nodes from *root* at the termination of the program, and the program terminates. The *parent* relation defines a tree because, except for *root* that does not have a parent, every node acquires a parent that has been marked earlier, and it never changes its parent. A formal proof of this fact needs to introduce level number for each node as an auxiliary variable. The root's level number is 0, and a child's level number is one more than its parent, so, the structure is a tree.

Breadth-first traversal can label the nodes with distinct numbers so that every node at level *i*+1 has a higher number than any at level *i* and nodes within a level are numbered in increasing order from left to right. In particular, a node has a higher

label than its parent. Start with a counter initialized to 0. Whenever the command “ $\text{parent}[v] := u$ ” is executed, assign the counter value to v and then increment the counter. I show this scheme explicitly in the program for the breadth-first traversal of a tree in Figure 6.24.

A breadth-first traversal scans every edge exactly once and spends $\Theta(1)$ time in scanning. So, its running time is $\Theta(m)$ for a graph with m edges. The additional space requirement is for the list $vroot$ and set $marked$. Prove the invariant that $vroot$ and $marked$ are disjoint as sets, and there are no duplicates in $vroot$. So, space requirement is $\Theta(n)$, where n is the number of nodes.

Breadth-first traversal of undirected graph The breadth-first traversal algorithm for undirected graph is almost identical to that for directed graph. In fact, the program given above works; every edge $u-v$ is examined twice, as incident edge of u and of v . The first examination occurs when, say, u is removed from $vroot$ and v is unmarked; then v acquires u as its parent. The next examination occurs when v is removed from $vroot$, but u is marked by then, so this edge has no further effect. Effectively, the undirected graph is treated as a directed graph where each undirected edge $u-v$ is replaced by two directed edges, $u \rightarrow v$ and $v \rightarrow u$.

Breadth-first traversal of rooted tree A tree is a graph, so all traversal algorithms for graphs are also applicable to trees. There are certain simplifications for trees because every node other than the root has exactly one incoming edge (from its parent). The set $marked$ need not be defined at all and $\text{succ}(u) - \text{marked}$ can be replaced by $\text{succ}(u)$; this is because every child of node u is unmarked when u is removed from $vroot$. And, of course, there is no need to identify the parent of a node, which is already given.

Breadth-First traversal of tree

```

vroot, ec := ⟨root⟩, 0;
while vroot ≠ ⟨⟩ do
  u, vroot := vroot.head, vroot.tail;
  bfn[u], ec := ec, ec + 1;
  for v ∈ succ(u) do
    vroot := vroot ++ ⟨v⟩
  endfor
enddo

```

Figure 6.24

The program in Figure 6.24 (page 295) assigns each node u a distinct breadth-first number $bfn(u)$. Variable ec is an *event counter* that starts at 0 and is incremented with the numbering of each node. The program needs no further explanation.

6.6.2 Depth-first Traversal

Depth-first traversal is perhaps the most important graph traversal technique. It induces a tree structure over the graph, like breadth-first traversal, that is exploited in several graph algorithms.

The abstract reachability algorithm of Figure 6.20 (page 291) can be specialized to obtain a depth-first traversal algorithm. Implement $vroot$ as a stack, so $u \leftarrow vroot$ returns the top item of the stack as u and the items in $succ(u)$ are added to the top of the stack (in arbitrary order unless there is a specified order over the successors). Thus, the scheme is inherently recursive, and that is how I describe it below, starting with the simple case of the traversal of a tree.

6.6.2.1 Depth-first Traversal of Tree

Depth-first traversal of a tree first visits its root, then each of its subtrees recursively. If the tree has just one node, the root, then the traversal just visits that node. The order in which the subtrees are traversed is in the order of the children of the root; if the children are unordered then the subtrees are traversed in arbitrary order.

The *preorder number* of a node is the order in which it is visited, starting with 0 for the root. The *postorder number* of a node is one more than the maximum postorder number of all its proper descendants; so it is assigned after completion of the traversal of all its subtrees.

I show an example of a traversal in Figure 6.25. Each node label is a symbol from the Roman alphabet, its preorder number precedes the label and the postorder

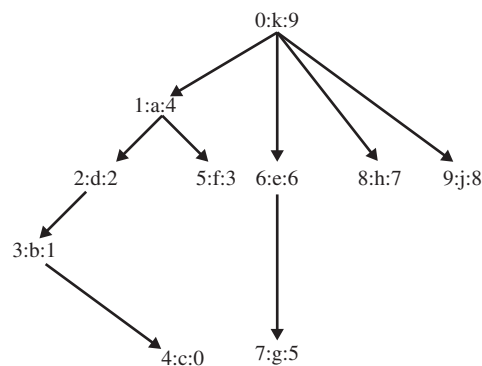


Figure 6.25 Depth-first traversal of a tree.

number follows the label. That is, the nodes are visited in the following sequence, given by consecutive preorder numbers: $\langle k a d b c f e g h j \rangle$. Note that c is followed by f because after visiting c the traversal of the left subtree of a is complete, and the traversal of its right subtree starts by visiting f .

Procedure $dfs(u)$ in Figure 6.26 traverses the subtree rooted at u in depth-first order. The preorder and postorder numbers of node u are, respectively, u_b and u_e (for *begin* and *end* of $dfs(u)$ procedure, below). The traversal employs two counters, ec_b and ec_e , initialized to 0 to record successive preorder and postorder values of nodes, respectively: u_b is set to ec_b when $dfs(u)$ starts and u_e to ec_e when $dfs(u)$ ends. During the execution of $dfs(u)$, every node v in its subtree is visited at some point; so, v_b and v_e are assigned values by the completion of $dfs(u)$.

Depth-first traversal of tree

```

Proc   $dfs(u)$ 
     $u_b, ec_b := ec_b, ec_b + 1;$ 
    for  $v$  in  $succ(u)$  do
         $dfs(v)$ 
    endfor ;
     $u_e, ec_e := ec_e, ec_e + 1$ 
endProc

{ Main program }
 $ec_b, ec_e := 0, 0;$ 
 $dfs(root)$ 

```

Figure 6.26

The **for** loop processes the children of u , $succ(u)$, in their designated order, or in arbitrary order if there is no specified order among the children.

6.6.2.2 Properties of Preorder and Postorder Numbering

From the incrementation of ec_b and ec_e by 1 each time when a preorder and postorder number is assigned, the values of ec_b and ec_e are monotone increasing. So the preorder and postorder numbers are consecutive starting at 0.

Terminology Henceforth, T_u denotes the set of nodes in the subtree rooted at node u . Call u to be *lower* than v (or v *higher* than u) if $u_e < v_e$. Any tree has the property that two subtrees are either disjoint or one is a subtree of the other, hence fully contained in it. Define nodes u and v to be *independent* if T_u and T_v are disjoint.

That is, $u \notin T_v$ and $v \notin T_u$; so, neither of the nodes is an ancestor of the other, in particular $u \neq v$. For independent u and v given $x \in T_u$ and $y \in T_v$, x and y are also independent because T_x and T_y are disjoint. ■

Proposition 6.1 The smallest preorder value and the largest postorder value in T_u are u_b and u_e , respectively.

Proof. Informally, during the execution of $dfs(u)$ variable u_b is assigned first and u_e assigned last among all v_b and v_e in T_u , and ec_b and ec_e are monotone increasing.

The specification needed for the proof is:

$$\{true\} \quad dfs(u) \quad \{(\forall v : v \in T_u, v \neq u : u_b < v_b, u_e > v_e)\}.$$

The formal proof uses the proof rule for procedure implementation (see Section 4.5.8, page 161). ■

The following proposition about the numbering of independent nodes is used extensively later in this chapter. It is easy to see because traversals of the trees rooted at independent nodes are disjoint, so one traversal completely precedes the other.

Proposition 6.2 Suppose x and y are independent, $u \in T_x$ and $v \in T_y$. Then,

$$(x_e < y_e) \equiv (u_e < v_e).$$

Proof. Independent nodes x and y have a common ancestor, namely *root*. So, they have a lowest common ancestor, say w . Node w has children x' and y' so that x is in $T_{x'}$ and y in $T_{y'}$ (x and y themselves could be x' and y' , respectively). Therefore $x \in T_{x'}$ and $u \in T_{x'}$, similarly, $y \in T_{y'}$ and $v \in T_{y'}$.

Suppose x' precedes y' in $succ(w)$. In the traversal algorithm, $dfs(x')$ is completed before $dfs(y')$ starts. So $(x_e < y_e) \wedge (u_e < v_e)$. And if y' precedes x' in $succ(w)$, $(y_e < x_e) \wedge (v_e < u_e)$. ■

Lemma 6.1 Convexity rule

Suppose $x_e \leq y_e \leq z_e$ and z is an ancestor of x . Then z is an ancestor of y .

Proof. If y is the same as x or z , then z is an ancestor of y . So, assume that y differs from both x and z , that is, $x_e < y_e < z_e$. Suppose y and z are independent. Then,

$$\begin{aligned} & y_e < z_e \\ \Rightarrow & \{y \text{ and } z \text{ are independent and } x \in T_z. \text{ Apply Proposition 6.2}\} \\ & y_e < x_e \\ \Rightarrow & \{\text{given: } x_e < y_e\} \\ & \text{false} \end{aligned}$$

So, y and z are dependent and, since $y_e < z_e$, z is an ancestor of y from Proposition 6.1. ■

A depth-first traversal of a tree scans every edge $u \rightarrow v$ exactly twice because the implementation of $dfs(u)$ both stacks and unstacks u once each time. So the running time is $\Theta(n)$ for a tree of size n . The additional space requirements are for the preorder and postorder numbers and the stack, which is $\Theta(n)$.

6.6.2.3 Depth-first Traversal of Directed Graph

The depth-first traversal algorithm for a directed graph, starting at a designated *root* node, is a generalization of the one for tree. The traversal constructs a *depth-first tree* over the nodes reachable from *root*. Each node u in the tree is assigned preorder and postorder numbers, u_b and u_e , respectively, during the construction of the tree. Choosing a different node as the *root* will yield a different tree.

A node that is a successor of several nodes is visited multiple times during the traversal. To avoid visiting such a node more than once, mark a node when it is visited for the first time (boolean $u.mark$ is *true* if u is marked), and ignore it if it is already marked. Also, to construct a depth-first tree we need to compute the parent of every node v in the tree, $v \neq root$; this information was already available when the graph was a tree as in Section 6.6.2.1 (page 296). In the case of a general graph, the parent of node v is the node from which v is visited for the first time. Let boolean $u.done$ signal the completion of $dfs(u)$, analogous to $u.mark$ that signals the start of $dfs(u)$.

Program **Depth-first traversal of directed graph** in Figure 6.27 (page 300) visits all nodes reachable from *root* in depth-first order and assigns them preorder and postorder numbers.

See Figure 6.28 for traversal of the graph in Figure 6.1(a) (page 267) with node k as the root. I show the preorder number preceding a node label and the postorder number following it. There are four kinds of edges: (1) a *tree* edge is directed from a parent to a child, shown as a solid arrow, (2) a *forward* edge is directed from an ancestor to a descendant, excluding from a parent to a child, shown as a finely dotted arrow, (3) a *back* edge is from a descendant to an ancestor, also shown as a finely dotted arrow, and (4) a *cross* edge is between two independent nodes, shown as a thick dashed arrow.

All the nodes are reachable from k , so the depth-first tree includes all nodes and edges of the graph. Had the traversal started at node a as the root it would have excluded the nodes unreachable from a , $\{h, j, k\}$. See Section 6.6.2.5 (page 302) for a strategy for visiting all nodes of a graph even when there is no node from which all nodes are reachable.

Depth-first traversal of directed graph

```

Proc dfs(u)
  ub, ecb, u.mark := ecb, ecb + 1, true;
  for v ∈ succ(u) do
    if v.mark then skip
    else
      parent[v] := u;
      dfs(v)
    endif
  endfor
  ue, ece, u.done := ece, ece + 1, true
endProc

{ Main program }
for u ∈ V do { V is the set of nodes in the graph }
  u.mark, u.done := false, false
endfor ;
ecb, ece := 0, 0;
dfs(root)

```

Figure 6.27

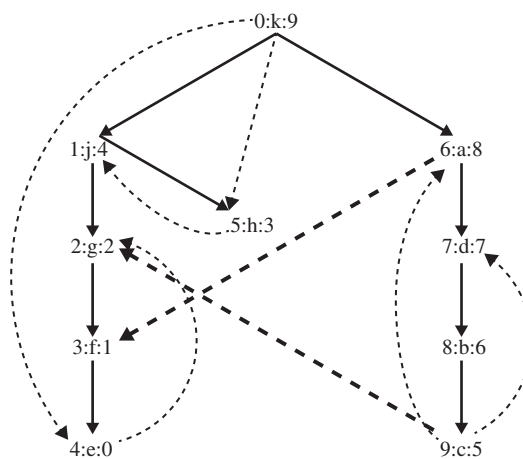


Figure 6.28 Depth-first traversal of the graph in Figure 6.1.

Terminology In subsequent discussions, the terms related to trees—parent, child, ancestor—refer to the depth-first tree, whereas successor, edge, path refer to the given graph.

6.6.2.4 Properties of Depth-first Traversal of Directed Graph

Proposition 6.3 DFS1

Execution of $dfs(u)$ terminates for all u .

Proof. The execution of $dfs(u)$ first decreases the set of unmarked nodes and then calls $dfs(v)$. Since each recursive call decreases the size of the set of unmarked nodes, termination is guaranteed. ■

Proposition 6.4 DFS2

The *parent* relation defines a tree.

Proof. Except for *root* that does not have a parent, every node acquires a parent when it is marked and never changes its parent. Further, the parent has been marked earlier, that is, has a lower preorder value, guaranteeing a tree structure. ■

Identifying different kinds of edges during depth-first traversal As I have remarked earlier, a depth-first traversal identifies each edge of a directed graph as one of four kinds: tree edge, forward edge, back edge or cross edge (see Figure 6.28). In the code of $dfs(u)$ if the test “if $v.mark$ ” fails, v will be marked and become a child of u , so $u \rightarrow v$ is a tree edge. If the test succeeds, that is, $v.mark$ already holds, then v has been marked earlier, and its type is determined by the relative values of the preorder and postorder numbers of u and v : (1) if $u_b < v_b$, then v was marked after u was marked, so it is a descendant u , and $u \rightarrow v$ is a forward edge, additionally, $v.done$ holds as well because $dfs(v)$ would have been completed; (2) if $u_b > v_b$, then v was marked before u was marked, so v is either independent of u ($v.done$ holds) or (3) its ancestor ($\neg v.done$ holds). These claims can be proved directly from the code of $dfs(u)$. I outline the needed invariant for the proof in Appendix 6.B.3 (page 376).

Edge-ancestor lemma and Path-ancestor theorem Edge-ancestor lemma, below, establishes an important property of depth-first traversal, that a cross edge is directed from a higher to a lower node based on postorder numbering. See Figure 6.28 in which all cross edges run from nodes with higher postorder numbers to lower numbered nodes: $(a : 8) \rightarrow (f : 1)$ and $(c : 5) \rightarrow (g : 2)$.

The Edge-ancestor lemma is given in the following contrapositive form that is more useful in formal derivations.

Lemma 6.2 Edge-ancestor

For any edge $u \rightarrow v$, $u_e < v_e \equiv v$ is ancestor of u . ■

I prove the lemma in Appendix 6.B.1 (page 373) from the annotated code of $\text{dfs}(u)$. An equivalent statement of the lemma is that for any edge $u \rightarrow v$: $u_e < v_e \equiv u \rightarrow v$ is a back edge.

Observe that absence of self loop is essential for this lemma. Given self loop $u \rightarrow u$, $u_e \not< u_e \Rightarrow u$ is not ancestor of u , violating the definition of ancestor.

Corollary 6.1 A cross edge is directed from a higher to a lower node. ■

The following theorem is a generalization of the edge-ancestor lemma. It captures an important property of depth-first traversal in directed graphs. It is proved in Appendix 6.B.2 (page 375).

Theorem 6.8 Path-ancestor

Given $u \xrightarrow{p} v$, where $u_e < v_e$, the highest node in p is the lowest common ancestor of all nodes in p . ■

The depth-first traversal of a directed graph scans every edge at most twice. So, its running time is $\Theta(m)$, where the number of edges is m . The additional space requirements are for the mark bits, preorder and postorder numbers and the stack, which is $\Theta(n)$ for a graph with n nodes.

6.6.2.5 Visiting All Nodes of the Graph: Augmented Graph

The depth-first traversal algorithm marks only the nodes reachable from *root*. Typically, not all nodes in a graph are reachable from a single root node. In that case, multiple depth-first traversals are used, first starting at one node as *root*, then choosing another node that has not been visited as *root* for the next depth-first traversal, and continuing until all nodes are visited. The resulting depth-first forest is often loosely called a depth-first tree of the graph.

The multiple traversal scheme is formalized in Figure 6.29. At any stage, *unmarked* is the set of all unmarked nodes. The root node is chosen from *unmarked* for each traversal. The process is repeated as long as there is an unmarked node.

Depth-first traversal of all nodes of a graph

```

{ unmarked = the set of all nodes in the graph }
while unmarked  $\neq$  {} do
    root  $\in$  unmarked;
    dfs(root)
enddo

```

Figure 6.29

Augmented graph A simpler way to visit all nodes of a graph is to *augment* the graph by adding a fictitious root node, *ficRoot*. This node has no incoming edge and has outgoing edges to every other node, so that all nodes are reachable from *ficRoot*. Apply $dfs(ficRoot)$. Then remove *ficRoot* and all its outgoing edges. The resulting structure is a depth-first forest. The subtrees may have cross edges among them.

6.6.2.6 Depth-first Traversal of Undirected Graph

The depth-first traversal algorithm for directed graphs (Figure 6.27, page 300) is applicable to undirected graphs as well; treat each undirected edge $u-v$ as two directed edges, $u \rightarrow v$ and $v \rightarrow u$.

For any edge $u \rightarrow v$, where $u_e < v_e$, v is an ancestor of u from edge-ancestor lemma, Lemma 6.2 (page 301). So, the dual edge $v \rightarrow u$ is between an ancestor and a descendant. Therefore, there is no cross edge. Further, each forward edge has a corresponding back edge. It is customary to remove the forward edges. Then the remaining edges are from parent to child and back edges from descendants to ancestors.

6.6.3 An Application of *dfs*: Topological Order in Acyclic Directed Graph

A topological order of the nodes of an acyclic directed graph is a total order in which node u precedes v if $u \rightsquigarrow v$. Reverse topological order is the reverse of this order, that is, v precedes u if $u \rightsquigarrow v$. Depth-first traversal can determine if a directed graph is acyclic and compute a topological order. Note that there may be several topological orders for the same acyclic graph.

Figure 6.30(a) is a depth-first tree of the acyclic graph of Figure 6.7 (page 275), where the traversal started at a fictitious root, *ficRoot* (see Section 6.6.2.5 (page 302))

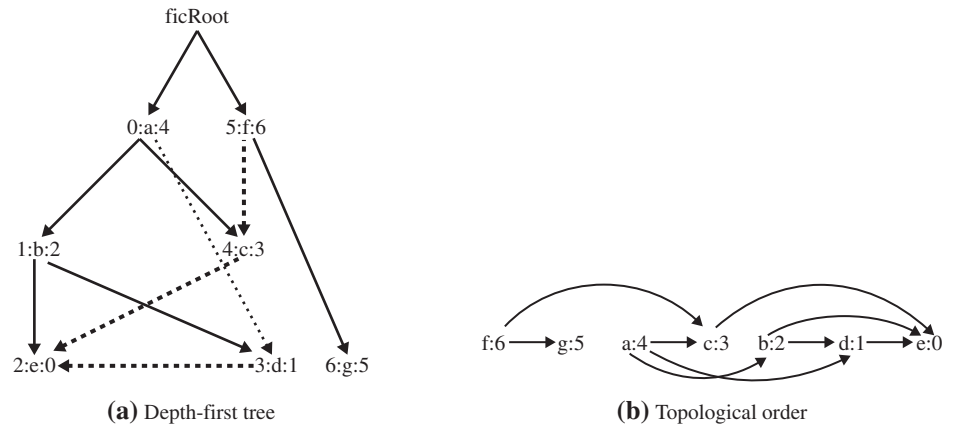


Figure 6.30 Topological sort of the acyclic graph in Figure 6.7.

for a discussion of augmented graph). The traversal order is given by the preorder numbers of the nodes. There is no back edge (that is no coincidence), one forward edge $a \rightarrow d$ shown as a fine dotted line, and three cross edges, $d \rightarrow e$, $f \rightarrow c$ and $c \rightarrow e$, shown as thick dashed line. The nodes are listed in decreasing postorder numbers in Figure 6.30(b). I will show that they are listed in a topological order.

Lemma 6.3 A directed graph is acyclic iff there is no back edge in a depth-first traversal.

Proof. I show that a graph has a cycle iff it has a back edge in a depth-first traversal.

(1) There is a back edge $u \rightarrow v$:

Then v is an ancestor of u , from the definition of back edge. So, there is a path, $v \rightsquigarrow u$, from v to u in the depth-first tree. Therefore, there is a cycle $u \rightarrow v \rightsquigarrow u$ in the graph.

(2) There is a cycle:

Let h be the highest node in the cycle and $u \rightarrow h$ its incoming edge in the cycle. Now $u_e < h_e$ because h is the highest node. Therefore, from the edge-ancestor lemma, (Lemma 6.2, page 301), h is an ancestor of u , or $u \rightarrow h$ is a back edge. ■

Topological sort From the edge-ancestor lemma, Lemma 6.2 (page 301), for any edge $u \rightarrow v$, $u_e < v_e$ iff $u \rightarrow v$ is a back edge, and from Lemma 6.3 (page 304) there is no back edge in an acyclic graph. Therefore, the decreasing postorder numbers of the nodes are in topological order. A list of nodes in this order can be compiled after a depth-first traversal. Or, during a depth-first traversal, add a node to the front of a list when it is assigned its postorder number so that an ancestor precedes a descendant in the list.

6.7 Connected Components of Graphs

6.7.1 Undirected Static Graph

Any reachability algorithm can be used to identify the connected components of an undirected graph. For any node u the set of its reachable nodes is a connected component. All reachable nodes from $root$, identified during the execution of $dfs(root)$, form a connected component. So, the program in Figure 6.29 (page 302) can identify the connected components, one for each invocation of $dfs(root)$. Another possibility is to do a depth-first traversal of the augmented graph, as described in Section 6.6.2.5 (page 302). The nodes in the subtree under each child of the fictitious root is a connected component.

6.7.2 Undirected Dynamic Graph: Union-Find Algorithm

A *dynamic graph* is one in which nodes and edges may be added and removed. A reachability algorithm that runs on a dynamic graph will have to contend with a new graph each time it is run. If the changes to the graph are limited, some of the data structures from one graph may be reused for the next run of the algorithm with the altered graph. In this section, I describe a classic algorithm, *union-find*, that checks for reachability in an undirected graph with a fixed set of nodes where edges can only be added to the graph, but not removed.

The problem is one of maintaining a dynamic equivalence relation. It has applications in many areas of computer science, such as computing equivalence of finite state machines and in type inference systems. I show one application in Section 6.10.3 (page 345) in finding a minimum spanning tree.

Problem description To motivate the problem, consider a road network that is being built to link various spots (nodes) in a community. The project manager is told every time a road (edge) linking two nodes is completed. At any point during the project, she may want to know if there is a route, sequence of roads, linking some pair of nodes. Therefore, the manager would like to have an algorithm that (1) implements a *union* command of the form “link x to y ” by adding edge $x-y$ when a road is completed and (2) answers queries of the form “is there a path between x and y ?” using operation *find*. See Figure 6.31 for a partially completed road network in which road $c-h$ is under construction.

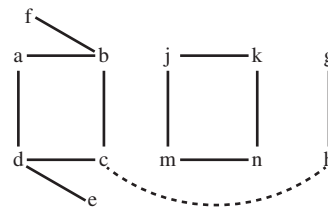


Figure 6.31 A road network under construction.

For a small project, it may suffice to add the edges on a map manually as they are being completed and do a visual check for the paths between designated nodes. For a large number of nodes—consider pathways in the brain as we discover new connections—a more systematic strategy is needed.

The edges of an undirected graph define an equivalence relation; nodes x and y are in the same equivalence class if a path $x \sim y$ exists. Figure 6.31 has the following equivalence classes: $\{a, b, c, d, e, f\}$, $\{j, k, m, n\}$, $\{g, h\}$. Upon completion of connection ch , subsets $\{a, b, c, d, e, f\}$ and $\{g, h\}$ will be joined to form a single equivalence class.

To answer the question “are x and y connected?”, check if their equivalence classes are identical. The problem is to devise efficient implementations of *union* and *find*.

6.7.2.1 Algorithm

There is a very efficient algorithm for this problem that implements each operation in near constant amortized time. There is no need to store the graph explicitly since our only interest is in storing the equivalence classes. Use a rooted tree over the nodes to store each equivalence class. All nodes of a tree belong to the same equivalence class and those in different trees belong to different ones. See Figure 6.32 for a tree representation of the equivalence classes in Figure 6.31.

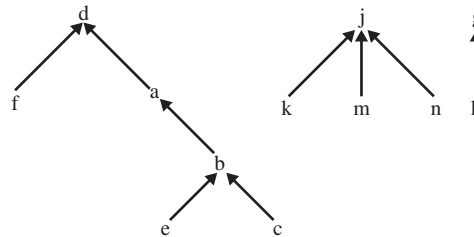


Figure 6.32 Tree representation of the graph of Figure 6.31.

The root of a tree is the unique representative of that equivalence class. The tree structure is stored by recording the parent of each node x , unless x is the root. Initially, before there is any edge in the graph, each node is in a distinct equivalence class, so each node is the root of a distinct tree.

The *find* operation on node x returns the root of its tree, by following the path $x \rightsquigarrow x'$ to root x' . That is, start at node x and repeatedly compute the parent of a node until the node has no parent; so it is the root. In Figure 6.32, the equivalence class of b is determined by following the path $b \rightarrow a \rightarrow d$ to root d . Two nodes are equivalent if the roots of the trees to which they belong are identical.

The *union* operation of x and y first performs two *finds* to locate the roots x' and y' of the trees to which nodes x and y belong. If $x' \neq y'$, then combine the two trees by adding the edge $x' \rightarrow y'$ or $y' \rightarrow x'$, that is, designating x' as the parent of y' , or conversely. Figure 6.33 shows the trees after doing a union of c and h in Figure 6.32; here g is the root of the combined tree.

An array of node ids, $A[0..n]$, can implement this algorithm efficiently where each node has a distinct id within $0..n$. Store the id of the parent of i in $A[i]$ if i is not a root. For a root node j , it is customary to store j at $A[j]$. The representation is both space efficient in using $\mathcal{O}(n)$ storage for all trees and time efficient in traversing a path to the root.

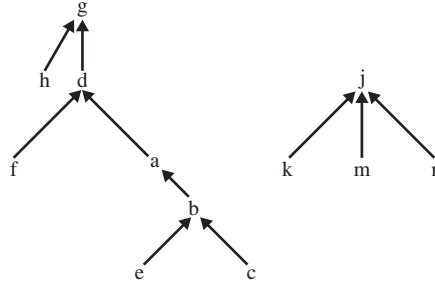


Figure 6.33 After completing the connection $c-h$ of Figure 6.31.

6.7.2.2 Optimization Heuristics

I present two heuristics, one for union and the other for find, that make this algorithm outstandingly efficient.

Optimization heuristic for Union; join by rank The union operation applied to two nodes first determines their roots, x and y . I have not prescribed which of x and y becomes the parent of the other. Intuitively, we should try to keep the trees as balanced as possible so there are no long paths. Let the *size* of a node be the number of nodes in its subtree. To keep the tree balanced, one obvious approach is to make x the parent of y (insert $y \rightarrow x$) if the size of x is larger than the size of y .

Actually, there is a better heuristic called *union by rank*.² Every node is initially assigned a rank 0, that is, when every tree is a singleton. The rule for union of trees with roots x and y is: (1) if x has a higher rank than y , make x the parent of y ; both x and y retain their current ranks, (2) if x and y have equal ranks, choose either node to be the parent of the other and increase the rank of the parent node by 1.

A possible sequence of unions that creates the trees in Figure 6.34 is (the parent node appears first in a pair): $\langle (j, k) (j, m) (g, h) (j, n) (d, f) (b, e) (d, a) (b, c) \rangle$; the rank of a node is shown next to the node name.

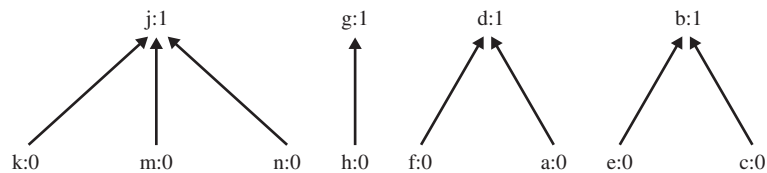


Figure 6.34 Nodes with ranks.

2. Either heuristic (by size or by rank) gives the same time bound. There is a slight difference in the space requirement: $\mathcal{O}(\log \log n)$ for rank and $\mathcal{O}(\log n)$ for size.

Next, suppose the tree with roots d and b are to be joined. They have equal ranks. Break the tie arbitrarily by choosing d as the root and increasing the rank of d (see Figure 6.35).

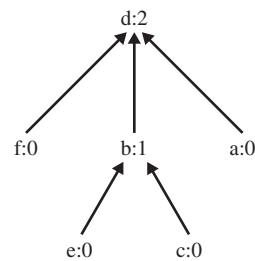


Figure 6.35 Union of d and b of Figure 6.34.

The ranks of the non-root nodes are not used in the algorithm. But they are essential for cost analysis (see Section 6.7.2.4, page 310).

Optimization heuristic for Find: path compression The next heuristic is called *path compression*. A find operation follows a path p from node x to its root x' , that is, $x \xrightarrow{p} x'$. After locating x' it follows path p again and for every node z in the path, except x' , changes its parent to x' . Every node that was in p now has a path of length 1 to the root, until the current root becomes a non-root. This heuristic speeds up subsequent executions of find. Note that the ranks are left unchanged during this operation. The extra pass over the path doubles the execution time of find, but the cost is made up by the efficiency gained in future finds.

Figure 6.36(a) shows the effect of compressing the path from c to d of Figure 6.35. Figure 6.36(b) shows the result of union of the tree in Figure 6.36(a) with the tree in Figure 6.34 (page 307) that has root g .

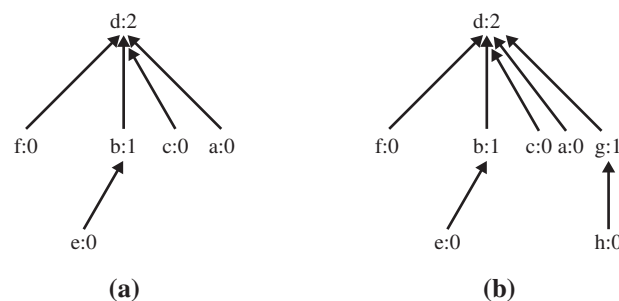


Figure 6.36 Path compression and union by rank.

Terminology The operation $adopt(x, y)$ for roots x and y makes y the parent of x (y adopts x as its child). The union operation then consists of two finds followed by a possible adopt. This terminology allows us to consider the costs of find and adopt operations separately.

6.7.2.3 Properties of Ranks

Proposition (6.5) lists a number of properties of ranks of nodes. All parts of the proposition, except part (6), can be proved easily by induction on the number of operations: assume that the property holds before a union or find operation and show that it holds after completion of the operation. I prove part (6)

Proposition 6.5

1. The rank of a node never decreases.
2. A non-root remains non-root and its rank remains constant.
3. The parent of a node has a higher rank.
4. After applying path compression to $x \xrightarrow{p} y \rightarrow z$, every node in p except y acquires a parent of higher rank.

This is because, from part (3), z has a higher rank than any node in p . And every node in p , except y , acquires z as its new parent.

5. A root node of rank j has at least 2^j descendants.

Note that this claim does not hold for non-roots. For example, if the path $h \rightarrow g \rightarrow d$ is compressed in Figure 6.36(b), g will have rank 1 and just one descendant, itself.

6. There are at most $n/2^j$ nodes of rank j or higher:

Let r_j be the number of nodes of rank j or higher and des_j the set of descendants of roots that have ranks j or higher. I prove the following invariant:

$$r_j \cdot 2^j \leq |des_j|, \text{ for all } j, j \geq 0. \quad (\text{RI})$$

Since $|des_j| \leq n$, conclude that $r_j \cdot 2^j \leq n$, or $r_j \leq n/2^j$.

Invariant RI holds initially because each node has rank 0, and there are n nodes, so $r_0 = n$. Further, each node is a root with itself as its only descendant, so $r_0 \cdot 2^0 \leq n$.

A find operation does not change any root, rank or descendant of any root; so, it preserves RI. For $adopt(x, y)$, every node acquires a root of the same or higher rank after the operation, from item (4) of this proposition; so, the right side of RI never decreases. If x and y have unequal ranks before $adopt(x, y)$, no node changes rank after the operation; so, the left side of RI is unchanged, and RI holds. I next consider the case of equal ranks of x and y in $adopt(x, y)$. Suppose x and y have ranks of k before $adopt(x, y)$. Then y 's rank is $k+1$ after the operation. RI continues to hold for all ranks other than $k+1$. For $k+1$ since r_{k+1} increases by 1, the left side of RI increases by 2^{k+1} . The right side, des_{k+1} ,

increases in size by at least 2^{k+1} because: (1) before $adopt(x, y)$ the descendants of x and y , which were disjoint, did not belong to des_{k+1} because their roots had rank k , (2) after the operation all of them belong to des_{k+1} because their root, y , has rank $k + 1$, and (3) there are at least 2^{k+1} such descendants because each of x and y had at least 2^k descendants, from item (5) of this proposition. Hence, RI is preserved.

6.7.2.4 Amortized Cost Analysis

The given algorithm is extremely efficient, but not on a per operation basis. A single find may incur a cost of $\log n$; so a naive cost analysis makes it a $\mathcal{O}(m \cdot \log n)$ algorithm for n nodes and m operations. But the combined cost of m operations, the amortized cost, is nearly $\mathcal{O}(m)$; so the per operation cost is nearly constant. I say “nearly” because the actual amortized cost is $\mathcal{O}(m \cdot \alpha(n))$, where α , known as the inverse of double Ackermann function, grows so slowly that it does not exceed 5 unless n exceeds the number of atoms in the universe. See the analysis in [Tarjan and van Leeuwen \[1984\]](#) and [Fredman and Saks \[1989\]](#).

I prove a slightly worse bound of $\mathcal{O}(m \cdot \log^* n)$ in this section. Function $\log^*$, called the iterated logarithm, grows extremely slowly so that the “atoms in the universe” remark also applies to it. However, it is a much faster growing function than α , a galloping horse compared to a snail.

Iterated logarithm In this section I use base 2 for $\log$ without making it explicit.

Informal definition of $\log^*$ is given by: $\log^* n = i$ where $\overbrace{\log \log \cdots \log(n)}^i \leq 1$. That is, $\log^* n$ is the number of times $\log$ has to be applied to n so that the function value becomes less than or equal to 1. Formally,

$$\log^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\lceil \log n \rceil) & \text{if } n > 1 \end{cases}$$

A dual function is defined by the sequence $t_0 = 0$ and $t_{i+1} = 2^{t_i}$. Then $\log^*(t_i) = i$. A *span* is a half-open interval; $span_i$ is $(t_i, t_{i+1}]$, that is, it excludes t_i and includes t_{i+1} . The number of spans within $(0, n]$ is $\log^* n$. I list some spans and the values of $\log^*$ for the numbers within those spans in [Table 6.2](#).

The highest value in $span_5, t_6$, is larger than the number of atoms in the universe. So, no practical problem will ever use a value in the next span, $span_6$.

Computing the amortized cost Each adopt operation has a cost of one unit, so the cost of m adopt operations is $\mathcal{O}(m)$. To calculate the costs of the find operations, I assign the unit cost of a single edge traversal, from a node x to its parent p , to one of three categories: (1) *fixed* cost if x or p is a root, (2) *out-span* cost if x and p

Table 6.2 Spans and iterated logarithms

i	$span_i = (t_i, t_{i+1}]$		$span_i$	$\log^* n$ for $n \in span_i$
0	$(0, 2^0]$	=	$(0, 1]$	0
1	$(2^0, 2^1]$	=	$(1, 2]$	1
2	$(2^1, 2^2]$	=	$(2, 4]$	2
3	$(2^2, 2^{2^2}]$	=	$(4, 16]$	3
4	$(2^{2^2}, 2^{2^{2^2}}]$	=	$(16, 65536]$	4
5	$(2^{2^{2^2}}, 2^{2^{2^{2^2}}}]$	=	$(65536, 2^{65536}]$	5

have ranks in different spans, and (3) *in-span* cost if x and p have ranks in the same span.

- fixed cost: A unit cost is incurred if x or p is a root; so this cost is $\mathcal{O}(m)$ for m operations.

- out-span cost: A unit cost is incurred if the rank of parent p is in a higher span than that of x . There are $\log^* n$ spans; so the out-span cost for each find operation is at most $\log^* n$ and for all m operations is $\mathcal{O}(m \cdot \log^* n)$.

- in-span cost: A unit cost is incurred if the rank of p and x are in the same span. Note that x is non-root since it has a parent; so its rank remains constant after it became a non-root, from Proposition 6.5 part (2). Further, from Proposition 6.5 part (3), the rank of p is higher than that of x ; also the next parent of x has a higher rank than that of p , from Proposition 6.5 (4). So, x acquires parents of ever higher ranks every time the edge from x to its parent is traversed. Suppose the rank of x is in $span_i$. The in-span cost of traversing the edges from x to its parents over all operations is at most the length of $span_i$, which is at most t_{i+1} .

Rank of x is at least t_i because $span_i = (t_i, t_{i+1}]$. From Proposition 6.5 part (6), the number of such nodes is at most $n/2^{t_i} = n/t_{i+1}$. Therefore, the total in-span cost over all $span_i$ nodes is at most $t_{i+1} \times n/t_{i+1} = n$. For $\log^* n$ spans, the total in-span cost is $\mathcal{O}(n \cdot \log^* n)$.

Adding the fixed, out-span and in-span costs the algorithm has $\mathcal{O}(m \cdot \log^* n)$ amortized cost for m operations over n nodes.

6.7.3 Strongly Connected Components of Directed Graph

An excellent example of the usefulness of depth-first traversal is in identifying the strongly connected components of a directed graph. I describe an ingenious algorithm due to Kosaraju,³ which was independently discovered by Sharir [1981].

3. Unpublished manuscript dated 1978.

Algorithm outline For a given graph G , its inverse graph G^{-1} is obtained by reversing the directions of the edges of G . Observe that a pair of nodes are strongly connected in G iff they are strongly connected in G^{-1} . So, the strongly-connected components of G and G^{-1} are identical. Assign the postorder numbers from a depth-first traversal of G to the nodes of G^{-1} .

Theorem 6.9 Let r be the highest node in G^{-1} , and R the set of reachable nodes from r in G^{-1} . Then R is a strongly connected component.

Proof. First, I show that every u in R is strongly connected to r , that is, $u \rightsquigarrow r \rightsquigarrow u$ exists in G :

r has a higher postorder number than u	r is the highest node
r is an ancestor of u in G	from Theorem 6.8 (page 302)
$r \rightsquigarrow u$ exists in G	from above
$u \rightsquigarrow r \rightsquigarrow u$ in G	$u \in R$, so $r \rightsquigarrow u$ exists in G^{-1} , i.e.
	$u \rightsquigarrow r$ in G

It follows that every pair of nodes in R are strongly connected through r . Further, since R is the set of reachable nodes from r in G^{-1} , no node outside R is reachable from r in G^{-1} . So, R is a strongly connected component. ■

This theorem suggests a simple procedure to construct the strongly connected components, $C_0, C_1, \dots, C_n$, in sequence. First compute R , as above, starting with the highest numbered node, and call it C_0 . Then remove all the nodes of C_0 and their incident edges from G^{-1} . Repeat the same procedure: identify the highest numbered node in the modified G^{-1} , and compute C_1 similarly. This process continues until the modified graph has no node.

Figure 6.37(a) shows the inverse graph, G^{-1} , of graph G of Figure 6.1 (page 267). The nodes are labeled with postorder numbers in the depth-first traversal of G , taken from Figure 6.28 (page 300). Figure 6.37(b) shows the strongly connected components of Figure 6.28 within boxes. First, the set of reachable nodes from k in G^{-1} , the highest node, is computed to be $\{k\}$, a strongly connected component. Next, node k (and its incident edges) are removed from the graph. The highest node now is a and its set of reachable nodes $\{a, b, c, d\}$ forms the next strongly connected component. This process is repeated successively with j and g to locate the next two strongly connected components, $\{j, h\}$ and $\{g, e, f\}$.

Implementation The algorithm, outlined above, can be implemented in a variety of ways. Any algorithm, including breadth-first or depth-first traversal, can be used to identify the reachable nodes in each modified version of G^{-1} . I outline an implementation, based on a single depth-first traversal of G^{-1} , that computes all the strongly connected components.

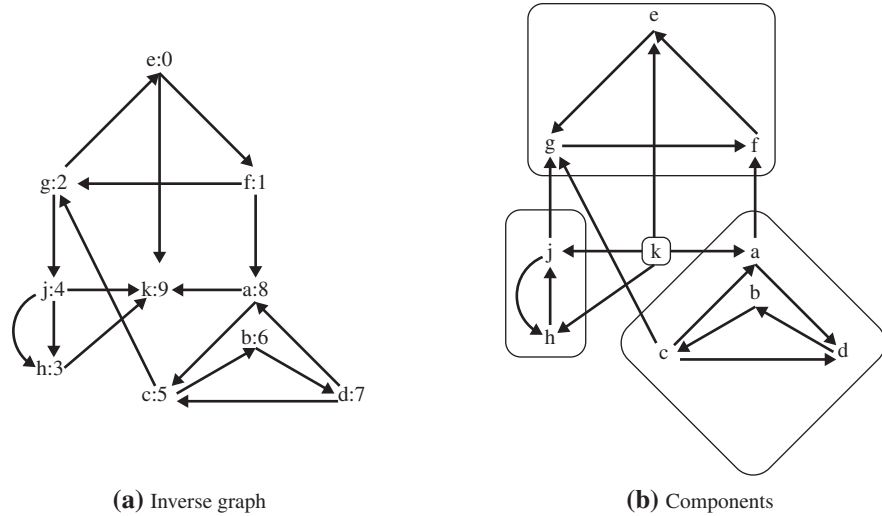


Figure 6.37 Inverse graph of Figure 6.1; strongly connected components of G .

First, augment G^{-1} with a fictitious root, *ficRoot*, as described in Section 6.6.2.5 (page 302). Then order all remaining nodes as $\text{succ}(\text{ficRoot})$ in *decreasing order of their postorder numbers in G* . This is an essential requirement to ensure that higher numbered nodes are traversed earlier. There is no requirement on the order of the successors of the remaining nodes. Do a depth-first traversal of G^{-1} from *ficRoot*. On completion of the traversal, remove *ficRoot* and its outgoing edges to obtain the depth-first subtrees. Each subtree's nodes form a strongly connected component.

Each part of the strongly connected component algorithm, two depth-first traversals, each run in $\mathcal{O}(m)$ time where m is the number of edges. So, the whole algorithm is linear in the number of edges.

6.7.3.1 Condensed Graph

It is often useful to treat all strongly connected nodes alike by merging all nodes of a strongly connected component to a single node (see “Merging nodes” in Section 6.2.6.2, page 272). This results in a *condensed* graph (or condensed component graph) of a directed graph. See Figure 6.38 for the condensed graph corresponding to Figure 6.28 (page 300). Each box has become a single node, named with the nodes that were merged to form the condensed node. The edges between nodes of various boxes are replaced with edges among the condensed nodes.

The condensed graph is useful in applications where the nodes within a strongly connected component need not be distinguished (see an example in Section 6.7.4).

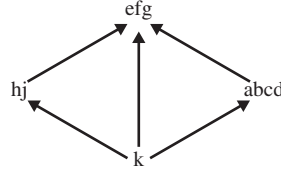


Figure 6.38 Strongly connected components (in boxes) of Figure 6.1.

A condensed graph is acyclic. This is because no two nodes in different strongly connected components are strongly connected, from the definition of strongly connected component. This acyclic graph is often scanned in topological order (u precedes v if $u \rightsquigarrow v$) or reverse topological order (v precedes u if $u \rightsquigarrow v$) (see Section 6.6.3). The sequence of components $\langle C_0, C_1, \dots, C_n \rangle$ in depth-first traversal are in reverse topological order because no edge is directed from C_i to C_j , $i < j$.

6.7.4 An Application: 2-Satisfiability

An important problem in computer science is to devise an efficient algorithm to decide the satisfiability of a boolean expression given in Conjunctive Normal Form (CNF), a conjunction of disjuncts (see Section 2.5.4, page 44). Expression f , below, is in CNF.

$$f :: (\neg p \vee q \vee r) \wedge (p \vee \neg r) \wedge (\neg q \vee r) \wedge (\neg p \vee \neg q \vee \neg r) \wedge (p \vee q \vee r)$$

If the number of literals in each disjunct is at most 3, as it is in f , the boolean expression is in 3-CNF and deciding its satisfiability is 3-satisfiability or 3-SAT. It has been a long-standing open problem to devise a polynomial time algorithm for the 3-SAT problem. In this section, I show a simple, linear time algorithm for the much simpler 2-SAT problem, where each disjunct has at most two literals [see [Aspvall et al. 1979](#)]. The 2-SAT problem has no apparent connection to graph theory. Yet, the problem is solved quite simply by appealing to graph algorithms, in particular for strongly connected component and topological sort. The algorithm decides not only satisfiability, but in case the boolean expression is satisfiable, it provides an assignment of truth values to variables that satisfies the boolean expression. The algorithm runs in linear time in the size of the boolean expression, the number of variables plus the number of conjuncts.

6.7.4.1 Reduction of 2-CNF Boolean Expression to a Graph

A disjunct in a 2-CNF expression is of the form $p \vee q$, where each of p and q is a literal, that is, a variable or its negation. Rewrite the boolean expression by replacing each disjunct, $p \vee q$, by $(\neg p \Rightarrow q) \wedge (\neg q \Rightarrow p)$, so the expression becomes a conjunction

of implications. Even though the two conjuncts are equivalent, both are needed in the following construction.

Next, construct a graph where each literal is a node and each implication, say $x \Rightarrow y$, is represented by edge $x \rightarrow y$. Each node x has a *dual* node $\neg x$ and each edge $x \rightarrow y$ a dual edge $\neg y \rightarrow \neg x$. Observe that the dual of the dual of a node/edge is the same node/edge.

The satisfiability problem is to assign truth values to the nodes so that (1) dual nodes have opposite values and (2) for each $x \rightarrow y$, $x \Rightarrow y$ holds. Taking *false* to be lower than *true*, each edge $x \rightarrow y$ is monotone, that is, the value of x is less than or equal to that of y , so that $x \Rightarrow y$ holds. Since implication is transitive, a satisfying assignment of truth values creates only monotone paths.

Example 6.1 Consider the 2-CNF:

$$(a \vee \neg e) \wedge (a \vee c) \wedge (\neg a \vee c) \wedge (\neg a \vee d) \wedge (b \vee e) \wedge (\neg b \vee \neg e) \wedge (\neg c \vee \neg d).$$

Transform the boolean expression using implication to get:

$$\begin{aligned} & (\neg a \Rightarrow \neg e) \wedge (e \Rightarrow a) \wedge (\neg a \Rightarrow c) \wedge (\neg c \Rightarrow a) \wedge (a \Rightarrow c) \wedge (\neg c \Rightarrow \neg a) \\ \wedge & (a \Rightarrow d) \wedge (\neg d \Rightarrow \neg a) \wedge (\neg b \Rightarrow e) \wedge (\neg e \Rightarrow b) \wedge (b \Rightarrow \neg e) \\ \wedge & (e \Rightarrow \neg b) \wedge (c \Rightarrow \neg d) \wedge (d \Rightarrow \neg c). \end{aligned}$$

The corresponding graph is shown in Figure 6.39.

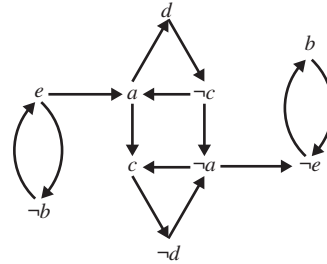


Figure 6.39 Converting 2-CNF to graph.

6.7.4.2 Graph Structure Determines Satisfiability

Theorem 6.10 A boolean expression is satisfiable iff each node and its dual are in different strongly connected components.

Proof. For strongly connected nodes x and y , there are paths $x \rightsquigarrow y$ and $y \rightsquigarrow x$. Since both paths are monotone, x and y have equal values in a satisfying assignment. Therefore, all nodes in a strongly connected component have equal value. Since a node and its dual have different values, they do not belong to the same strongly connected component.

Next, I show that if each node and its dual are in different strongly connected components, there is a satisfying assignment. Construct a condensed graph, as given in Section 6.7.3.1 (page 313), where each strongly connected component is represented by a single node. Observe that the dual of a strongly connected component is also strongly connected; so it is a condensed node. Section 6.7.4.3 gives a procedure for assigning truth values to the condensed nodes, which is the same value that is assigned to each node, that is, literal, in that component. ■

The condensed graph corresponding to graph of Figure 6.39 is shown in Figure 6.40, where each box represents a strongly connected component.

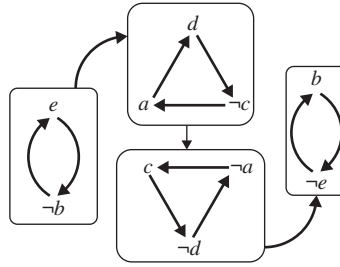


Figure 6.40 Strongly connected components of Figure 6.39.

6.7.4.3 Truth-value Assignment

The condensed graph is acyclic (see Section 6.7.3.1, page 313). Construct a topological order, $<$, over the nodes of the condensed graph. For Figure 6.40, let $c < b < e < d$ be the order, where I have chosen a single representative node from each component. The same order applies to all pairs of nodes from different components, so for p and q in different components where $p \rightarrow q$, we have $p < q$.

I summarize the various relations we have on hand: (1) implication, for example, $p \Rightarrow q$, introduced in converting a formula in CNF to a conjunction of implications over its literals, (2) directed edges in the graph, for example, $p \rightarrow q$, which represents the given implication, and (3) a topological order, for example, $p < q$, over the literals. Their relationships are given by: $p \Rightarrow q$ iff $p \rightarrow q$, and if $p \rightarrow q$, then $p < q$.

Apply the following rule to assign truth values to nodes in the condensed graph: If $p < \neg p$, then assign *false* to p and *true* to $\neg p$. In Figure 6.40, the rule leads to the following assignment: *false* to $\{a, b, c, \neg d\}$ and *true* to $\{a, \neg b, \neg c, d\}$.

I show that the rule for truth-value assignment is valid: if $p \rightarrow q$ is in the graph then $p \Rightarrow q$ holds. This requirement is trivially met if p is *false*. I show that if p is *true*, then q is also *true*.

$$\begin{array}{l}
\neg q \\
< \{p \rightarrow q \text{ exists; so, } \neg q \rightarrow \neg p \text{ exists and } \neg q < \neg p\} \\
\neg p \\
< \{p \text{ is true. From the truth-value assignment rule, } \neg p < p\} \\
p \\
< \{p \rightarrow q\} \\
q
\end{array}$$

From $\neg q < q$, using the truth-value assignment rule, q is *true*.

6.7.4.4 Satisfiability Algorithm

It is straightforward to combine some of the algorithms given previously to construct a linear time algorithm for 2-satisfiability. Construction of the graph from a given boolean expression requires a constant number of steps for each disjunct. The strongly connected component algorithm of Section 6.7.3 (page 311) runs in linear time. As a bonus, the components $C_0, C_1, \dots, C_n$, $n \geq 0$, are in reverse topological order. Assign truth values to the components in order from C_0 through C_n : *true* to each C_i , unless it already has a value, and *false* to its dual. This assignment satisfies the two required properties: (1) each node and its dual receive different values and (2) any two strongly connected nodes receive the same value.

6.8 Transitive Closure

The transitive closure of a graph, directed or undirected, tells us if a path exists between any pair of nodes. For undirected graphs, the problem is quite easy. Identifying the components of the graph yields the transitive closure: all nodes within a component are reachable only from each other. So, the problem is of interest mainly for directed graphs.

Given the adjacency matrix A of a directed graph, so that $A[u, v] \equiv u \rightarrow v$, its transitive closure A^+ satisfies $A^+[u, v] \equiv u \rightsquigarrow v$ and the reflexive and transitive closure A^* satisfies $A^*[u, v] \equiv (u = v \vee u \rightsquigarrow v)$; so A^+ includes paths of positive length and A^* includes zero-length paths as well. Observe that $A^+[u, u] = \text{false}$ means u does not belong to any cycle, so if $A^+[u, u] = \text{false}$ for all u , then the graph is acyclic.

Computing the transitive closure of a graph is important in many applications. A communication network can be represented by a directed graph where cables connect the adjacent nodes. The transitive closure determines the pairs of nodes that are connected by a path of cables. A variation of it determines the shortest paths between every pair of nodes in a network where edge lengths are specified (see Section 6.8.3, page 321).

As I write this, the spread of Coronavirus requires computing a transitive closure. Imagine a very large graph whose nodes are individuals and each edge records a contact between two individuals over a certain time period, that is, $u-v$ denotes that u has made physical contact with v in the last three weeks, say. If it is later discovered that certain individuals are infected, we can identify, using transitive closure, the identities of all individuals who have contacted an infected person directly, or indirectly through other individuals. The contact relation in this example yields an undirected graph, but we can imagine situations where the contact is asymmetric, yielding a directed graph.

6.8.1 Transitive Closure and Matrix Multiplication

The notion of transitive closure is related to closure of relations defined in Section 2.3.5 (page 31). I defined the transitive closure of binary relation A over set S as follows: if $x_0 A x_1, x_1 A x_2, \dots, x_{j-1} A x_j$, then $x_0 A^+ x_j$, for any $j, j \geq 1$. More formally, A^+ is the smallest relation that includes A and is transitive. The reflexive and transitive closure of A , A^* , is the smallest relation that includes A and is both reflexive and transitive.

Matrix multiplication Transitive closure can be expressed in terms of matrix multiplication. The following discussion uses elementary matrix theory. In particular, for matrices A and B of the same dimension, with typical elements $A[u, v]$ and $B[u, v]$, their sum $A + B$ is defined by the sum of the corresponding elements: $(A + B)[u, v] = A[u, v] + B[u, v]$. We restrict ourselves to matrix multiplication over square matrices of the same dimension. Given matrices $A[0..n, 0..n]$ and $B[0..n, 0..n]$,

$$(A \times B)[u, w] = (+v : 0 \leq v < n : A[u, v] \times B[v, w]), \text{ for } 0 \leq u, w < n.$$

For transitive closure of A with boolean elements, addition (+) is logical or ($\vee$) and multiplication ($\times$) logical and ($\wedge$). Let I be the identity matrix, that is, $I[u, u] = \text{true}$ and $I[u, v] = \text{false}$, for all u and $v, u \neq v$. Write A^i as the i -fold multiplication of A with itself, so $A^0 = I$, and $A^{i+1} = A \times A^i$, for $i \geq 0$. Then, $A^1 = A$. Observe that $A^{i+1}[u, v]$ is *true*, where $i \geq 0$, iff there is a path with i intermediate nodes from u to v .

There is a path between a pair of nodes iff there is a path of length smaller than n between these nodes, where n is the number of nodes in the graph. This is because any simple path can have at most n nodes (hence $n - 1$ edges). So, $A^+ = (+i : 1 \leq i < n : A^i)$. And, $A^* = (+i : 0 \leq i < n : A^i)$, which includes paths of length zero.

In order to compute A^* , compute each term $A^i, 0 \leq i < n$, and then add all the terms. Compute the terms in order, multiplying A with a term to get the next

term. A better procedure is to use the identity given in Exercise 9 (page 365), part (4), $A^+ = (+j : 0 \leq 2^j < n : A^{2^j})$ that gives a $\Theta(n^3 \cdot \log n)$ algorithm. Warshall's algorithm of Section 6.8.2 is far superior, needing only $\Theta(n^3)$ time.

It can be shown that transitive closure and (a single) matrix multiplication have the same time complexity. Theoretically, this is a better time bound than Warshall's algorithm, though Warshall's is simpler to program, and is better in practice in its running time.

6.8.2 Warshall's Algorithm

A $\Theta(n^3)$ algorithm for computing the transitive closure was invented by Warshall [1962]. First, number the nodes consecutively from 0 to $n - 1$ where the graph has n nodes. I write $u \xrightarrow{<t} v$ to mean that there is a path of non-zero length from u to v and all *intermediate* nodes in this path are numbered below t . The algorithm computes a sequence of matrices, $\langle W_0, \dots, W_t, \dots, W_n \rangle$, where $W_t[u, v] \equiv u \xrightarrow{<t} v$.

From the definition of W_t , $W_0 = A$, the adjacency matrix of the graph, because the only path from u to v with intermediate nodes numbered below 0 is just the edge $u \rightarrow v$, if it exists. And, $W_n[u, v]$ holds if there is any path p , $u \xrightarrow{p} v$, because all intermediate nodes in p are numbered below n . That is, $W_n = A^+$. For computing W_{t+1} from W_t , consider any two nodes u and v :

$$\begin{aligned}
 & W_{t+1}[u, v] \\
 \equiv & \{ \text{definition of } W_{t+1}[u, v] \} \\
 & u \xrightarrow{<t+1} v \\
 \equiv & \{ \text{case analysis: node } t \text{ is not on the path or it is} \} \\
 & u \xrightarrow{<t} v \vee (u \xrightarrow{<t+1} v \text{ and node } t \text{ is on this path}) \\
 \equiv & \{ \text{for the second term consider only simple paths} \} \\
 & u \xrightarrow{<t} v \vee u \xrightarrow{<t} t \xrightarrow{<t} v \\
 \equiv & \{ \text{rewrite using the definition of } W \} \\
 & W_t[u, v] \vee (W_t[u, t] \wedge W_t[t, v])
 \end{aligned}$$

The equation $W_{t+1}[u, v] = W_t[u, v] \vee (W_t[u, t] \wedge W_t[t, v])$ provides a simple procedure for computing A^+ : compute in sequence $\langle A = W_0, W_1, \dots, W_n = A^+ \rangle$, as described in the algorithm shown in Figure 6.41, below.

The outer **for** loop is executed in order of increasing values of t ; the inner loop is executed in arbitrary order for all matrix elements. At termination $A^+ = W_n$. Compute A^* by setting $W_n[u, u]$ to *true* for all u .

Each matrix element is computed in constant time, so each matrix W_t in $\Theta(n^2)$ time and the transitive closure in $\Theta(n^3)$ time.

Warshall's algorithm for transitive closure, A^+

```

 $W_0 := A;$ 
for  $t \in 0..n$  do
  for  $(u, v) \in \{(i, j) \mid i \in 0..n, j \in 0..n, i \neq j\}$  do
     $W_{t+1}[u, v] := W_t[u, v] \vee (W_t[u, t] \wedge W_t[t, v])$ 
  endfor
endfor

```

Figure 6.41**6.8.2.1 Space Requirement for Warshall's Algorithm**

As described, the algorithm needs storage space for n matrices. However, the computation of W_{t+1} requires only W_t . So, it is easy to modify the algorithm so that only two matrices need to be stored. In fact, the algorithm can be implemented with just one matrix W that plays the role of W_t for all t . Initially, $W := A$. Replace the command

$$W_{t+1}[u, v] := W_t[u, v] \vee (W_t[u, t] \wedge W_t[t, v]) \text{ by}$$

$$W[u, v] := W[u, v] \vee (W[u, t] \wedge W[t, v]).$$

The correctness of this modification is not obvious because either none, one or both of $W[u, t]$ and $W[t, v]$ may have been updated *during* this iteration before execution of the command $W[u, v] := W[u, v] \vee (W[u, t] \wedge W[t, v])$. However, note that

$$W_{t+1}[u, t] = W_t[u, t] \vee (W_t[u, t] \wedge W_t[t, t]) = W_t[u, t].$$

Similarly, $W_{t+1}[t, v] = W_t[t, v]$. Therefore, it is immaterial if these values have been updated.⁴ We can assert,

$$\{(\forall x, y :: W[x, y] = W_t[x, y])\}$$

$$W[u, v] := W[u, v] \vee (W[u, t] \wedge W[t, v])$$

$$\{W[u, v] := W_{t+1}[u, v]\}$$

Therefore, upon completion of the iteration for $t = k$, $(\forall x, y :: W[x, y] = W_{k+1}[x, y])$.

I show a different proof of correctness that uses a weaker invariant. It is applicable to a wider variety of problems, including for the implementation of the Bellman–Ford algorithm (Section 6.9.6.3, page 337). Define,

$$P_t :: (\forall x, y :: x \overset{t}{\rightsquigarrow} y \Rightarrow W[x, y] \Rightarrow x \rightsquigarrow y).$$

4. I am grateful to Greg Plaxton for this observation.

I show in Exercise 10 (page 366) that P_t is an invariant of the main loop. Here I show that invariance of P_t is sufficient to prove the correctness of Warshall's algorithm. First, after the initialization, $W := A, P_0$ holds. And, after termination of the iteration with $t = n - 1, P_n$ holds. So, at termination $x \stackrel{<n}{\rightsquigarrow} y \Rightarrow W[x, y] \Rightarrow x \rightsquigarrow y$ holds for each x and y . Since $x \stackrel{<n}{\rightsquigarrow} y \equiv x \rightsquigarrow y$, $W[x, y] \equiv x \rightsquigarrow y$ at the end of the program, as required.

6.8.3 All-pair Shortest Path Algorithm

This variation of Warshall's algorithm, due to Floyd [1962], computes the shortest paths among all pairs of nodes in a directed graph. The adjacency matrix element $A[u, v]$ is the length of edge $u \rightarrow v$. Edge lengths are non-negative, $A[u, u] = 0$ for all u and $A[u, v] = +\infty$ if there is no edge $u \rightarrow v$. Length of a path is the sum of the lengths of all its edges. Exercise 12 (page 367) shows that the requirement of non-negative edge lengths can be replaced by the weaker requirement that there is no cycle of negative length.

Redefine $W_t[u, v]$ as the length of the shortest path from u to v in which all *intermediate* nodes are numbered strictly below t . Then $W_0 = A$, and $W_n[u, v]$ is the desired shortest path length from u to v . The following derivation is similar to the one in Section 6.8.2, and the algorithm is a simple modification of Warshall's algorithm. For any two nodes u and v , not necessarily distinct:

$$\begin{aligned}
 & W_{t+1}[u, v] \\
 = & \{ \text{case analysis on paths: } t \text{ is not on the path or it is} \} \\
 & \min(\text{over all } u, v:: \text{path lengths over all } u \stackrel{<t}{\rightsquigarrow} v, \\
 & \quad \text{path lengths over all } u \stackrel{<t}{\rightsquigarrow} t \stackrel{<t}{\rightsquigarrow} v) \\
 = & \{ \text{rewrite using the definition of } W \} \\
 & \min(W_t[u, v], W_t[u, t] + W_t[t, v])
 \end{aligned}$$

Computing the shortest paths along with their lengths To compute the actual shortest path, store the penultimate node of the path whenever a shortest path length is updated. Thus, if the shortest path from u to v is $u \rightarrow v$, then store u , and if it is $u \rightsquigarrow x \rightarrow v$, then store x as the penultimate node for v . The actual shortest path to v , $u \rightsquigarrow v$, is the concatenation of the successive penultimate nodes from v backwards until u .

6.8.4 Compiling a Metro Connection Directory

A city metro system has a number of stations connected by a light-rail system. Trains run over the light-rails. There are several *lines* where each line runs over



Figure 6.42 Metro map of Washington, DC. Image courtesy of Washington Metropolitan Area Transit Authority © 2018.

a certain sequence of stations. For example, the metro system in Washington, DC, has six lines, each identified by a different color: Blue, Green, Orange, Red, Silver and Yellow (see Figure 6.42). A rider boards a train of line b (for Blue), say, at any station where the Blue line stops and she may ride the train to any other station

where that line stops. A line may not stop at all stations through which it runs. Riders connect from one line to another at a station where both lines stop.

The route map is typically given by a color-coded graph. Assume that the map has been converted to an adjacency matrix A that has a row and a column for each station. Matrix element $A[u, v]$ is a *set* of line names where $x \in A[u, v]$ means line x stops at both u and v , and runs *without a stop* from u to v .

Henceforth, L is the set of line names. Each route from u to v is a string over L , say rbg , denoting that it is possible to go from u to v first using Red (r) line, then transferring to Blue (b) line at some station and finally transferring to Green (g) line at some other station and continuing on that line to reach v eventually. This is also denoted by $u \xrightarrow{rbg} v$. There may be other ways of traveling from u to v , such as $u \xrightarrow{ry} v$ and $u \xrightarrow{sbg} v$.

It is convenient to have matrix A^+ where $A^+[u, v]$ is the set of *all* routes with the fewest transfers.⁵ For example, given that $u \xrightarrow{ry} v$ and $u \xrightarrow{sbg} v$ are the only routes between u and v , $A^+[u, v]$ will include only the shortest route ry . In particular, matrix A^+ will tell a rider if it is possible to go from u to v using a *single* line so that no transfer would be required, which is useful to many riders with disabilities.

The problem can be posed as a transitive closure problem and solved efficiently using Warshall's algorithm. This problem requires an application of the all-pair shortest paths algorithm (Section 6.8.3, page 321) in which all shortest paths, that is, the shortest strings for the routes, are retained. Trivially, $A^*[u, u]$ is $\{\varepsilon\}$.

It should be clear that a string like $rbrg$, in which r is repeated, is never in $A^*[u, v]$ because the rider can cover this route using rg by staying on the r line until the transfer to g , which is a shorter path. That is, a shortest path never has repetition of a symbol.

6.8.4.1 Definitions of $\times$ and $+$ for the Metro Directory

Each operator, $\times$ and $+$, has two sets of strings as operands, and it computes a single set of strings as the value. Operator $\times$ takes the Cartesian product of its operand sets by concatenating every pair of strings from its two operands. Each resulting string, say by concatenating rbg and rs to get $rbgrs$, gives a route, though this is not a shortest route. Operator $+$ takes the union of its operand sets and retains only the shortest strings.

5. The route itself may not be unique. Given $u \xrightarrow{r} w \xrightarrow{s} v$ and $u \xrightarrow{r} w' \xrightarrow{s} v$, $A[u, v]$ records only rs without indication about two possible ways of traveling over these lines.

6.8.4.2 Marshall's Algorithm for Metro Connection Directory

As in the algorithm of Figure 6.41, number the stations from 0 to $n - 1$. Variable $W_t[u, v]$ is a set of strings over L , where $x \in W_t[u, v]$ means that x is a shortest route (in the number of transfers) for travel from u to v using only the lines in x in order, and stopping only at the stations numbered below t . Clearly, $W_0 = A$ and $W_n = A^+$. Derivation of $W_{t+1}[u, v]$ follows the same pattern as before, using the operators $+$ and $\times$:

$$\begin{aligned} & W_{t+1}[u, v] \\ \equiv & \{ \text{definition of } W_{t+1}[u, v], \text{ using case analysis} \} \\ & W_t[u, v] + (W_t[u, t] \times W_t[t, v]) \end{aligned}$$

6.8.5 Transitive Closure Over Semiring

This section covers advanced material. The problems considered in Sections 6.8.2 (page 319), 6.8.3 (page 321) and 6.8.4 (page 321) are very similar in structure though the corresponding adjacency matrices have different kinds of elements. For computing the existence of paths in a graph (Section 6.8.2), each matrix element is a boolean value denoting the presence or absence of an edge. For the all-pairs shortest path problem (Section 6.8.3), the elements are non-negative numbers. For computing metro routes in Section 6.8.4, the elements are sets of strings. The sum and product operators required for matrix multiplication have different meanings for different problems: logical or ($\vee$) for sum and logical and ($\wedge$) for product in computing the existence of paths, minimum over numbers for sum ($\min$) and addition ($+$) for product for all-pairs shortest paths, and set union for sum and string concatenation for product for compiling the metro connection directory.

Transitive closure can be generalized to a much wider class of problems. The given examples are some of the instances. The theory uses the algebra of semiring. I do not cover the topic in detail in this book but give a quick sketch below.

Semiring A *semiring* has five components: $(E, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. Here E is a set of elements, $\oplus$ and $\otimes$ are binary operators (called *sum* and *product*) over E , and $\mathbf{0}$ and $\mathbf{1}$ are unit elements for $\oplus$ and $\otimes$, respectively. That is, $\mathbf{0} \oplus x = x = x \oplus \mathbf{0}$ and $\mathbf{1} \otimes x = x = x \otimes \mathbf{1}$. Further, $\mathbf{0}$ is annihilated by $\otimes$: $\mathbf{0} \otimes x = \mathbf{0} = x \otimes \mathbf{0}$.

There are several constraints on the operators: $\oplus$ is commutative and associative, $\otimes$ is associative, and $\otimes$ distributes over $\oplus$. Further, these properties carry over to infinite sums and products for a *closed* semiring, which is normally used in practice.

The elements of the adjacency matrix are elements of E ; equivalently, an edge of the graph has the corresponding matrix element as its label. The label of a path is the product, applying $\otimes$, of the labels of the edges along the path. This definition

also applies to infinite paths (that arise from infinite repetitions of cycles) since the product is defined for a closed semiring. The label on a path of length zero is $\mathbf{1}$. The value of $A^*[u, v]$ is the sum of the labels over all paths (applying $\oplus$) including the path of length zero, from u to v . Again, this sum may be over an infinite number of operands, which is defined for a closed semiring.

We can write an equation of the form given in Warshall's algorithm; one term in it requires the closure of a semiring element corresponding to infinite paths.

$$W_{t+1}[u, v] = W_t[u, v] \oplus (W_t[u, t] \otimes W_t[t, t]^* \otimes W_t[t, v]).$$

In the examples considered so far, $x^* = \mathbf{1}$ for all elements x of E . Then the right side of the equation reduces to $W_t[u, v] \oplus (W_t[u, t] \otimes W_t[t, v])$ and the computation is finite. In other cases, the closure computation is infinite. For example, consider the semiring $K = (Str, \cup, \cdot, \{\}, \{\varepsilon\})$, where Str is the set of all finite strings over some alphabet. The sum and product operators are set union and pairwise concatenation of strings over two sets of strings. The empty set and the set consisting of the empty string act as $\mathbf{0}$ and $\mathbf{1}$, respectively. Then the closure of a string s is the set of strings $\{\varepsilon, s, s.s, s.s.s, \dots\}$, so, the set is not finite.

6.9 Single Source Shortest Path

Consider a directed graph where each edge has a specified length. Finding a shortest path between two specified nodes in such a graph is of considerable importance in practice. Such paths are regularly used for packet routing in computer networks to minimize communication delays. Since computer networks are often huge in size and packet routing is used almost continuously, it is imperative to have a fast algorithm for this problem. There are many other problems that can be cast as shortest path problems: physical routing of traffic to minimize trip distance, congestion, power usage or the number of interchanges among highways, or to calculate degrees of separations from a root to a specific individual in a social network. The all-pair shortest path problem (Section 6.8.3, page 321) can be solved by repeated applications of a single source shortest path algorithm, though such a computation is usually slower than the one in Section 6.8.3 for dense graphs.

A $\Theta(n^2)$ algorithm when edge lengths are non-negative is given in Section 6.9.4 (page 328). The algorithm in Section 6.9.6 (page 335) handles negative edge lengths provided there are no negative cycles; it runs in $\Theta(n^3)$ time.

6.9.1 Formal Description of the Problem

Given is a directed graph with a single *source* node, henceforth called *root*. Each edge $u \rightarrow v$ has an associated real number $e(u, v)$, possibly negative, as its length.

The length of a path is the sum of edge lengths along that path; a path with no edges has length 0.

See Figure 6.43 for a graph whose edges are labeled with their lengths. All edge lengths in this graph are positive, but edge lengths could also be negative.

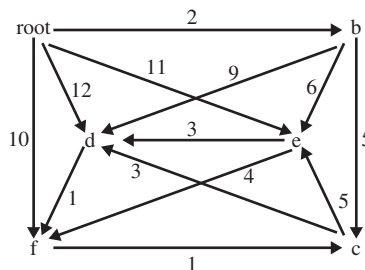


Figure 6.43 Graph edges are labeled with lengths.

If there is no path from *root* to some node *u*, then the shortest path length to *u* is ∞ . Given that there is a path to *u*, a shortest path exists iff *u* is not part of a cycle with negative total length. If it is part of such a cycle, then the length of the shortest path is $-\infty$ because the shortest path includes infinite repetitions of that cycle. Conversely, if all cycles of which *u* is a part have positive length, then a shortest path to *u* does not include any cycle, so, it is a simple path. Since there is a finite number of simple paths to any node, a shortest path exists. Cycles of length zero do not alter path lengths, so the same argument applies. Call a cycle of non-negative length a non-negative cycle.

Shortest path algorithms find the shortest path from *root* to *every* node in the graph. Finding the shortest path from *root* to just one specified node is no simpler than finding shortest paths to all nodes.

Terminology In this section a “path to node *u*” is a path from *root* to *u*. The *distance* of node *u*, $d(u)$, is the length of the shortest path from *root* to *u*. If there is no edge $u \rightarrow v$, its length is taken to be $+\infty$. The set of successor/predecessor nodes of *u* are $\text{succ}(u)/\text{pred}(u)$.

6.9.2 Shortest-path Equation

Theorem 6.11 A graph in which all cycles are non-negative satisfies the following equation.

$$\begin{aligned} d(\text{root}) &= 0, \text{ and} \\ d(v) &= (\min u : u \in \text{pred}(v) : d(u) + e(u, v)), \text{ for all } v, v \neq \text{root} \end{aligned} \quad (\text{SP-Eqn}).$$

Proof. Given that all cycles are non-negative, the shortest path to *root* is the empty path with length 0. For $v \neq \text{root}$, every path to *v*, including the shortest path, has to

include a predecessor of v , so it is of the form $root \xrightarrow{p} u \rightarrow v$ for some predecessor u of v . Length of the shortest such path is $(\min u : u \in pred(v) : d(u) + e(u, v))$. ■

Longest path The problem of finding the longest path is analogous; simply replace every edge length d by $-d$ and find the shortest path. The longest path to a node has infinite length if the node is on a cycle of positive length. If the graph is acyclic, see Section 6.9.3, the problem has a non-trivial solution.

6.9.3 Single Source Shortest Path in Acyclic Graph

Perhaps the easiest problem to solve in this context is one for an acyclic graph. An acyclic graph has no cycle, hence all cycles are non-negative. Apply (SP-Eqn): compute $d(u)$ for all predecessors u of v before computing $d(v)$. This is easily done for an acyclic graph because the nodes can be listed in topological order (see Section 6.6.3, page 303) so that $u \in pred(v)$ means u precedes v in this order.

In Figure 6.44, $td[u]$ is the tentative distance to u , initially ∞ for every non-root node⁶. Below, V is the set of all nodes in the graph and L the list of nodes in topological order.

Single source shortest paths in acyclic graph

```
 [root] := 0; for u ∈ V − {root} do td[u] := ∞ endfor ; for v ∈ L − {root} do { Process nodes in their order in L }     td[v] := (min u : u ∈ pred(v) : td[u] + e(u, v)) endfor { (∀u :: td[u] = d(u)) } |
```

Figure 6.44

6.9.3.1 Maximum Segment Sum

Given is a list of real numbers $\langle a_1 a_2 \dots a_n \rangle$. Any interval $i..j$ corresponds to a segment $\langle a_i \dots a_{j-1} \rangle$ whose sum is $(+k : i \leq k < j : a_k)$. It is required to find the segment having the maximum sum. The problem is interesting only because the numbers in the list could be positive, negative or zero. So, the maximum segment could be empty, with sum 0.

This problem is also treated in Section 7.4.3.2 (page 418) in connection with function generalization in recursive programming.

6. I write $d(u)$, where u is within parentheses, to denote a constant, and $td[u]$, where u is within square brackets, for a variable whose value may change during program execution.

The graph formulation is to convert the problem to a longest path problem over an acyclic graph. So the algorithm has linear cost in both time and space, but the space can also be optimized to a constant. Construct a graph with the nodes *source*, *sink* and $\{u_0, u_1, \dots, u_n\}$. The *source* node has an outgoing edge of length 0 to every u_i , $source \xrightarrow{0} u_i$, for $0 \leq i \leq n$. Each u_i has an outgoing edge of length a_i to u_{i+1} , $u_i \xrightarrow{a_{i+1}} u_{i+1}$ for $0 \leq i < n$, and an outgoing edge of length 0 to *sink*, $u_i \xrightarrow{0} sink$, $0 \leq i \leq n$.

The graph is acyclic, each path from *source* to *sink* represents a segment and the path length is the corresponding segment sum. The empty sequence is the path $source \xrightarrow{0} u_i \xrightarrow{0} sink$, for any i . The sequence of nodes along a longest path, excluding *source* and *sink*, is a maximum segment. So, the maximum segment can be computed by suitable application of the program in Figure 6.44 (page 327).

6.9.4 Shortest Paths with Non-negative Edge Lengths

I develop an important algorithm, due to Dijkstra [1959], that computes the shortest paths from a single *root* node to all other nodes in $\Theta(n^2)$ time, where n is the number of nodes and the edge lengths are non-negative. For the rest of this section, assume that edge lengths are positive real numbers; zero edge lengths are treated in Exercise 17 (page 368). Figure 6.45 shows a graph in which the shortest path, shown with dashed edges, from *root* to *d* is $root \xrightarrow{2} b \xrightarrow{5} c \xrightarrow{3} d$; it has length 10.

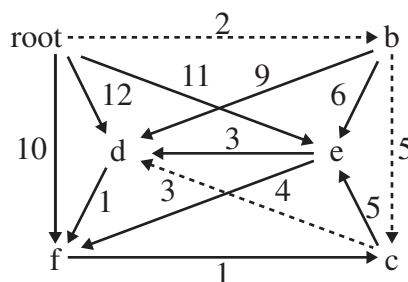


Figure 6.45 Shortest path $root \rightarrow b \rightarrow c \rightarrow d$ in dashed lines.

6.9.4.1 Mathematical Basis of the Algorithm

The relationship described in (SP-Eqn) (Section 6.9.3, page 327) is valid for general graphs. But it is not effective as a computational tool if the graph has cycles. To see this, consider a graph in which nodes x and y are predecessors of each other. In order to compute $d(x)$, (SP-Eqn) needs $d(y)$, and vice versa. This circularity is

overcome in Dijkstra's algorithm by assigning a distinct natural number, called a *rank*, to each node *in order of their distances*; the lower the rank the closer is the node to *root*. Clearly, *root* has the lowest rank, 0. Nodes at equal distances are ranked in arbitrary order. The algorithm identifies and computes the distances in order of ranks. The next observation is crucial to identifying ranks in increasing order and their distances.

Observation 6.3 Ranks of the nodes along any shortest path are monotone increasing.

Proof. The proof is by induction on the number of edges in the shortest path. For the empty path, from *root* to itself, the result follows vacuously. Let $root \xrightarrow{p} u \rightarrow v$ be a shortest path to *v*. Then *p* is a shortest path to *u*, otherwise it can be replaced by a shorter path to *u* that would result in a shorter path to *v*. So, $d(v) = d(u) + e(u, v)$ and, from $e(u, v) > 0$, $d(u) < d(v)$. Hence the rank of *u* is less than that of *v*. Inductively, the ranks are monotone along *p*. ■

6.9.4.2 Development of the Algorithm

Tentative path lengths The algorithm computes the ranks, and distances in order of ranks. At any point during the execution, the nodes of the first *k* ranks for some *k*, $0 \leq k$, and their distances are known. Call this set *ranked* and call a node ranked if it is an element of *ranked*. The remaining nodes are *unranked*. Initially, no node is ranked and finally all nodes are ranked.

Tentative distance $td[x]$ is the length of the shortest path to *x* in which *every node in the path preceding x is ranked*; $td[x]$ is ∞ if there is no such path. Initially *ranked* is an empty set, so there is no non-empty path of the required form for any node *x*. So, $td[x] = \infty$, for $x \neq root$, and $td[root] = 0$ because there is an empty path to *root* in which there is no preceding node.

Using Observation 6.3 and the definition of *td*, if *z* is ranked then $td[z] = d(z)$. Combine it with fact that $td[x]$ is the length of some path to *x* and $d(x)$ the length of the shortest path, to get invariant I1:

I1 :: $td[z] = d(z)$, for every ranked node *z*, and $td[x] \geq d(x)$, for every node.

For any unranked node *y*, consider a path of the form $root \xrightarrow{p} x \rightarrow y$, where *x* is ranked and *p* is the shortest path to *x*. The length of this path is $d(x) + e(x, y) = td[x] + e(x, y)$. Since $td[y]$ is the minimum length over all such paths, we have invariant I2:

I2 :: for every unranked node *y*, $td[y] = (\min x : x \in ranked : td[x] + e(x, y))$.

Figure 6.46 shows the graph of Figure 6.43 in which the first two nodes in the ranking, *root* and *b*, are shown within the box. The number following a node label is the tentative distance to that node. For each ranked node, the tentative distance is

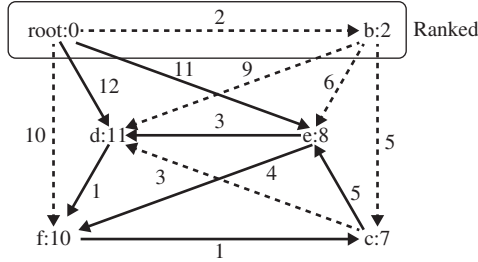


Figure 6.46 Two ranked nodes.

the actual distance, so the shortest path length to node b is 2. The shortest paths to unranked nodes using only ranked nodes as intermediate nodes are shown using dashed edges in Figure 6.46.

Identifying the next node in ranking

Observation 6.4 The unranked node that has the smallest td value is the next node in the ranking.

Proof. Suppose v is the unranked node that is the next in ranking. Then, for any unranked y ,

$$\begin{aligned}
 & td[v] \\
 = & \{ \text{from invariant } I1 \} \\
 & d(v) \\
 \leq & \{ v \text{ is the next node in ranking} \} \\
 & d(y) \\
 \leq & \{ \text{from invariant } I1 \} \\
 & td[y]
 \end{aligned}$$

■

Figure 6.47 shows that c is identified as the next node in ranking in Figure 6.46 because it has the least tentative distance. It has been added to the list of ranked nodes in Figure 6.47. Further, the tentative distances to the remaining unranked

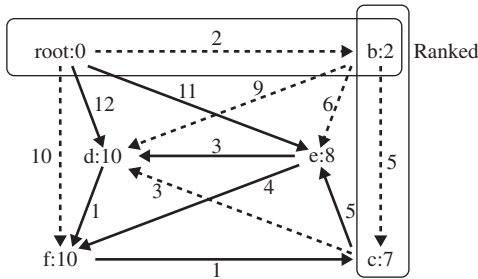


Figure 6.47 Three ranked nodes.

nodes are updated, as explained below. In particular, node d has a tentative distance of 11 in Figure 6.46 and 10 in Figure 6.47. ■

Updating $td[y]$ for all unranked y After v , the next node in ranking, is identified, it is moved from *unranked* to *ranked*. Then the algorithm has to reestablish $I2$ because v , as a newly ranked node, might affect $td[y]$ for some unranked y . Rewrite the expression for $td[y]$ from $I2$:

$$\begin{aligned}
 & (\min x : x \in \text{ranked} : td[x] + e(x, y)) \\
 = & \{ \text{rewrite the term for } x = v \text{ separately} \} \\
 & \min((\min x : x \in \text{ranked} - \{v\} : td[x] + e(x, y)), td[v] + e(v, y)) \\
 = & \{ \text{The first term is the current value of } td[y] \text{ because } v \text{ was unranked} \} \\
 & \min(td[y], td[v] + e(v, y))
 \end{aligned}$$

The second term, $td[v] + e(v, y)$, applies only if y is a successor of v . So, the update step following the identification of v as the next ranked node is:

for each unranked successor y of v , assign $td[y] := \min(td[y], td[v] + e(v, y))$.

Figure 6.48 is a partially annotated program where invariant I is the conjunction of $I1$ and $I2$. V is the set of nodes and $\text{unranked} = V - \text{ranked}$.

Figure 6.47 has the final distances for all the nodes, as can be checked by running the algorithm for the next two steps. The next node in ranking is e whose tentative distance of 8 is lower than that of d at 10. Then the remaining node d retains its tentative distance of 10.

Performance of the algorithm The algorithm executes n iterations. Each iteration computes $td[v]$, taking the minimum over at most n terms. In a straightforward implementation, this takes $\mathcal{O}(n)$ time. Thus, the total computation time for $td[v]$'s across all iterations is $\mathcal{O}(n^2)$. Each iteration also updates $td[y]$ for each unranked y . This update is done for $v \rightarrow y$ when v is the newly ranked node and y is unranked. So, across all iterations the total computation time for updates is $\mathcal{O}(m)$, where m is the number of edges. Since $m < n^2$, the computation time is dominated by $\mathcal{O}(n^2)$.

A different implementation uses a priority queue, q , which may be more efficient for sparse graphs. The items of q are the tuples $(td[y], y)$ for all unranked y . So, the size of q is at most n . To extract the tuple with the minimum value of $td[y]$ from q and then update q takes $\mathcal{O}(\log n)$ time; over all iterations the time taken is $\mathcal{O}(n \log n)$. Updating $td[y]$ amounts to replacing its current value by a lower value in q . This consumes $\mathcal{O}(\log n)$ time for each edge $x \rightarrow y$, where x is ranked and y unranked; across all iterations, $\mathcal{O}(m \log n)$ time is spent. So, the time complexity is $\mathcal{O}(m \log n)$, which is better than $\mathcal{O}(n^2)$ for sparse graphs.

Other data structures, such as balanced trees, have also been used for implementation.

Dijkstra's algorithm for single source shortest path

```

ranked := {}; td[root] := 0;
for x ∈ V − {root} do td[x] := ∞ endfor;
{ I :: I1 : (∀x : x ∈ ranked : td[x] = d(x)),
  I2 : (∀y : y ∈ unranked : td[y] = (min x : x ∈ ranked : td[x] + e(x, y))) }
while unranked ≠ {} do
  v ∈ {x | x ∈ unranked, td[x] = (min y : y ∈ unranked : td[y])};
  { v is the next node to be ranked }
  ranked := ranked ∪ {v};
  { (∀x : x ∈ ranked : td[x] = d(x)) }
  for y ∈ (succ(v) ∩ unranked) do
    td[y] := min(td[y], td[v] + e(v, y))
    { td[y] = (min x : x ∈ ranked : td[x] + e(x, y)) }
  endfor
  { I }
enddo
{ I, all nodes ranked }
{ (∀x :: td[x] = d(x)) }

```

Figure 6.48

Computing the shortest paths The given algorithm computes the distances but not the paths. The actual paths can be computed along with the distances using the technique given in Section 6.8.3. For each node *z*, store the penultimate node on the shortest path to it in *pen*[*z*].

From *I*2, *td*[*y*] = (min *x* : *x* ∈ *ranked* : *td*[*x*] + *e*(*x*, *y*)). And *pen*[*y*] is the ranked node *x* for which the minimum is attained. The move of *v* from *unranked* to *ranked* does not affect *pen*[*v*]. Initially, set *pen*[*z*] to *root* for all *z* except *root*. Modify the update step of *td*[*y*] as follows:

```

for y ∈ (succ(v) ∩ unranked) do
  if td[y] ≤ td[v] + e(v, y) then skip
  else td[y], pen[y] := td[v] + e(v, y), v
  endif
endfor

```

6.9.5 A Simulation-based Shortest Path Algorithm

An entirely different development of Dijkstra's algorithm uses refinement of a real-time concurrent program to a sequential program. No background in concurrent

programming is assumed in this section. The refinement is based on discrete event simulation; again no background in simulation is assumed.

6.9.5.1 Real-time Concurrent Program for Shortest Path

I sketch a fantasy algorithm next. Imagine that each node is a computer, and all computers share a common clock so they see the same time. Nodes communicate using light quanta, called *photons*, that are sent along the edges. A photon carries a *hop count* with it whose meaning will be explained shortly. The computation commences with the *root* receiving a photon with hop count 0 at time 0 from some external entity.

Whenever any node x receives (along an incoming edge) a photon with hop count n at time t , it records the triple (n, t, x) in a common *ledger* and sends photons with hop counts $n + 1$ along each of its outgoing edges. These computations take no time. A photon takes $e(x, y)$ time units to travel along $x \rightarrow y$. Observe that a typical graph with cycles may have an infinite number of paths to some nodes, so the procedure may never terminate and the ledger may become infinitely long. Initially, $(0, 0, \text{root})$ is the only ledger entry.

Correctness I prove that (n, t, x) is a ledger entry iff there is a path to x of length t with n edges; regard this assertion as the invariant of the algorithm.

The proof of this claim is by induction on n . For $n = 0$, the only path with 0 edges is the one from *root* to itself, which is given by the initial ledger entry $(0, 0, \text{root})$.

$$\begin{aligned}
 & \text{there is a path to } x \text{ of length } t \text{ with } n \text{ edges, where } n > 0 \\
 \equiv & \{ \text{graph theory} \} \\
 & \text{for some predecessor } y \text{ of } x: \\
 & \text{there is a path with } n - 1 \text{ edges of length } t - e(y, x) \text{ to } y \\
 \equiv & \{ \text{induction} \} \\
 & \text{for some predecessor } y \text{ of } x: \\
 & \text{there is a ledger entry } (n - 1, t - e(y, x), y) \\
 \equiv & \{ \text{step of the algorithm at } y \} \\
 & \text{for some predecessor } y \text{ of } x: \\
 & y \text{ receives a photon of hop count } n - 1 \text{ at } t - e(y, x) \\
 \equiv & \{ x \text{ receives a photon with hop count } n \text{ at } t, n > 0, \text{ iff} \\
 & \quad x \text{ has a predecessor } y \text{ that receives } n - 1 \text{ at } t - e(y, x) \} \\
 & \quad x \text{ receives a photon with hop count } n \text{ at } t \\
 \equiv & \{ \text{step of the algorithm at } x \} \\
 & \text{there is a ledger entry } (n, t, x)
 \end{aligned}$$

6.9.5.2 Simulation-Based Implementation

In the given algorithm, the nodes operate in real time where every node has access to a perfectly synchronized clock. Further, the nodes execute their steps in no time, only the passage of photons consume time. These are unrealistic assumptions for actual computer networks. However, the algorithm can be implemented on a single computer (uniprocessor) by using discrete event simulation. I describe the simulation scheme as it pertains to this algorithm.⁷

Call the receipt of a photon an *event*. Occurrence of an event may cause other events to happen in the future, in this case receipt of a photon may cause sending of photons that will cause receipts of photons, that is, events, in the future. A simulation is a sequential program that processes all events in order of their occurrences. Note that the ledger contains entries for all the events in order of their occurrences.

At any point during simulation an initial segment of the ledger would have been processed (or the corresponding events have *happened*). The remaining part of the ledger, events yet to be processed, is stored in a data structure called *event_queue*. Clearly, only a finite portion of the *event_queue* would be known. These are the scheduled events that are guaranteed to happen in the future, caused by events that have already happened.⁸

Henceforth, the hop count, which was introduced to simplify the proof, is dropped because the number of edges in the shortest path is irrelevant in this formulation. The ledger contains only entries of the form (t, x) , where t is the length of the shortest path to x , so there is at most one such entry for any x . The other ledger entries are stored in *event_queue* which contains a tuple (t, x) , to denote that node x will receive a photon at time t , that is known to happen but has not yet happened.

Initially, the only known event is $(0, \text{root})$. A step of simulation removes the earliest event, that is, the entry (t, x) in the *event_queue* that has the smallest time component t (break ties arbitrarily), and processes it as follows. If event (t, x) corresponds to the receipt of the first photon by x , then (1) insert (t, x) in *ledger* because the length of the shortest path to x is t , and (2) since x sending a photon to y causes the event $(t + e(x, y), y)$ to happen in the future, add such events for all successors y of x to *event_queue*. If (t, x) corresponds to the receipt of a non-first photon by x , ignore it.

7. Direct implementations of concurrent real-time algorithms are not common. Author's research group has designed and implemented language Orc, see <https://orc.csres.utexas.edu/>, in which this algorithm is coded exactly as described, in about 6 lines.

8. The shortest path computation is a special case in simulation where an event cannot be canceled. That is, a photon cannot be recalled just because its sending node decides to cancel the send. In general simulation, scheduled events may be canceled, so a more elaborate definition of an event in *event_queue* is needed, but the simulation algorithm remains about the same.

To determine if an event corresponds to the receipt of the first photon by a node, call a node *lit* if it has already received a photon, *unlit* otherwise (arrival of a photon lights up a node, which persists). Every entry (t, x) in the ledger denotes that the distance of x is t and x is *lit*. Every entry (t, y) in *event_queue* denotes that y will receive a photon at t .

Simulation algorithm for single source shortest path

```

{ every node is unlit }
ledger := {};
event_queue := {(0, root)} ∪ {(∞,  $x$ ) |  $x \neq \text{root}$ };
while “there is an unlit node” do
    “remove  $(t, x)$  from event_queue, where  $t$  is the smallest value for any unlit  $x$ ”;
    {  $x$  is unlit;  $x$  receives its first photon at  $t$  }
    “light up  $x$ ” {  $x$  is now lit };
     $d(x) := t$ ;
    for “every unlit successor  $y$  of  $x$ ” do
        “add  $(t + e(x, y)), y$  to event_queue”
    endfor
enddo

```

Figure 6.49

Relationship to Dijkstra’s algorithm The algorithm of Figure 6.49 is identical to Dijkstra’s algorithm of Figure 6.48 except for refinement of its data structures. The predicates *lit* and *unlit* are implemented by the sets *ranked* and *unranked*, respectively. Both *ledger* and *event_queue* are implemented by array *td*; ledger entry (t, x) denotes that $d(x) = t$.

6.9.6 Shortest Paths with Negative Edge Lengths

In this section, I present an algorithm, due to Bellman [1958] and Ford Jr. [1956], for the single source shortest path problem where edge lengths are allowed to be negative. The algorithm computes distances to all nodes if there is no negative cycle, or detects the existence of a negative cycle.

Dijkstra’s algorithm is restricted to edge lengths being non-negative. In the vast majority of cases this is a valid assumption. There are, however, some network protocols that have to contend with negative edge lengths. An example is an electric car that consumes considerable amount of energy going uphill, a small amount

going on flat roads and regenerates energy going downhill. The edge lengths specify energy consumption; so, some of them are negative. The shortest desired path between two nodes is the one consuming the least energy.⁹

6.9.6.1 Underlying Equations

Let $d_i(v)$ be the length of the shortest path from *root* to v that uses at most i edges. The following equations, (BF1–BF3), allow iterative computations of $d_i(v)$ for all nodes v up to any value of i .

$$d_0(\text{root}) = 0 \quad (\text{BF1})$$

$$d_0(v) = \infty, \text{ for all } v, v \neq \text{root} \quad (\text{BF2})$$

$$d_{i+1}(v) = \min(d_i(v), (\min u : u \in \text{pred}(v) : d_i(u) + e(u, v))), \text{ for all } v \quad (\text{BF3})$$

Equations (BF1) and (BF2) are easy to see: the empty path to *root* has length 0, and there is no empty path to any non-root node, so, $d_0(v) = \infty$ for such nodes. For (BF3), let $u \xrightarrow{\leq i} v$ denote that there is a path from u to v of at most i edges. The paths to v of at most $i + 1$ edges have either (1) at most i edges, whose minimum length is $d_i(v)$, or (2) exactly $i + 1$ edges of the form $\text{root} \xrightarrow{\leq i} u \rightarrow v$, for some predecessor u of v . The minimum over such a path is $d_i(u) + e(u, v)$, and over all such paths is $(\min u : u \in \text{pred}(v) : d_i(u) + e(u, v))$. (BF3) follows.

6.9.6.2 Negative Cycles

Equations (BF1–BF3) apply to graphs with arbitrary edge lengths; so they can be used for iterative computations for graphs with negative cycles. I derive the termination condition for the iterative computation: in a graph of n nodes there is no negative cycle iff $d_{n-1}(v) = d_n(v)$ for all v , and in the absence of negative cycles $d_{n-1}(v) = d(v)$, the distance to v .

Theorem 6.12 All cycles are non-negative iff $d_{n-1}(v) = d_n(v)$, for all v .

Proof. proof is by mutual implication.

- Suppose all cycles are non-negative. Then the shortest path to any node is a simple path with at most $n - 1$ edges, so, $d_{n-1}(v) = d(v)$ for all v .

Next, I show $d_n(v) = d(v)$ for all v . This holds for $v = \text{root}$. For any $v, v \neq \text{root}$:

$$\begin{aligned} & d_n(v) \\ = & \{\text{from equation (BF3)}\} \\ & \min(d_{n-1}(v), (\min u : u \in \text{pred}(v) : d_{n-1}(u) + e(u, v))) \\ = & \{d_{n-1}(v) = d(v) \text{ for all } v\} \end{aligned}$$

9. There is a story about politicians in a country who promised to connect all its towns using roads that go only downhill. Driving always generates energy!

$$\begin{aligned}
& \min(d(v), (\min u : u \in \text{pred}(v) : d(u) + e(u, v))) \\
= & \{(\text{SP-Eqn}), \text{Section 6.9.2 (page 326)}\} \\
& \min(d(v), d(v)) \\
= & \{\text{property of min}\} \\
& d(v)
\end{aligned}$$

- Suppose $d_{n-1}(v) = d_n(v)$ for all v . I show that all cycles are non-negative. Consider any cycle C in which the predecessor of node v is v' .

$$\begin{aligned}
& d_n(v) \\
\leq & \{\text{equation (BF3) with } n-1 \text{ for } i \text{ and } v' \text{ for } u\} \\
& d_{n-1}(v') + e(v', v) \\
= & \{\text{from the assumption, } d_{n-1}(v') = d_n(v')\} \\
& d_n(v') + e(v', v)
\end{aligned}$$

Taking the sum of $d_n(v) \leq d_n(v') + e(v', v)$ over all v in C :

$$\begin{aligned}
& (+v : v \in C : d_n(v)) \leq (+v : v \in C : d_n(v') + e(v', v)) \\
\Rightarrow & \{\text{rewriting}\} \\
& (+v : v \in C : d_n(v)) \leq (+v : v \in C : d_n(v)) + (+v : v \in C : e(v', v)) \\
\Rightarrow & \{\text{Canceling terms from both sides}\} \\
& 0 \leq (+v : v \in C : e(v', v))
\end{aligned}$$

That is, the sum of edge lengths of C is non-negative. ■

6.9.6.3 Implementation

The implementation of the Bellman-Ford algorithm directly follows the given derivation; assign $d_0(v)$ and then iteratively compute $d_{i+1}(v)$ up to $d_{n-1}(v)$, for all v . Then check for negative cycle using Theorem 6.12. We can save storage by using a single variable $td[v]$ to store the successive values of $d_i(v)$, as we did for Warshall's algorithm in Section 6.8.2.1 (page 320).

In Figure 6.50, the invariant of the outer **for** loop, following all the initializations, is I , as shown in the annotation.

$$I :: (\forall v :: td[v] = d_i(v)).$$

6.9.6.4 Cost Analysis

The given algorithm executes n iterations in its outer loop. The inner loop examines each edge from a predecessor u of v exactly once in the command

$$td[v] := \min(td[v], (\min u : u \in \text{pred}(v) : td[u] + e(u, v))).$$

Bellman-Ford algorithm for single source shortest path

```

{ V is the set of nodes }
 [root] := 0; for x ∈ V − {root} do td[x] := ∞ endfor; for i ∈ 0..n − 1 do   { I }   for v ∈ 0..n do     { ((∀x : 0 ≤ x < v : td[v] = di+1(v)), (∀x : v ≤ x < n : td[v] = di(v))) }     td[v] := min(td[v], (min u : u ∈ pred(v) : td[u] + e(u, v)))     { ((∀x : 0 ≤ x ≤ v : td[v] = di+1(v)), (∀x : v < x < n : td[v] = di(v))) }   endfor   { (∀v :: td[v] = di+1(v)) } endfor ; { (∀v :: td[v] = dn−1(v)) } { if dn−1(v) = d(v), there is no negative cycle. Note, td[v] = dn−1(v) } if (∀v :: td[v] = (min u : u ∈ pred(v) : td[u] + e(u, v)))   then { there is no negative cycle }     for v ∈ V do d(v) := td[v] endfor   else “report there is a negative cycle” endif |
```

Figure 6.50

Therefore, the running time is $\mathcal{O}(m \cdot n)$ for a graph with n nodes and m edges. For a dense graph m could be $\Omega(n^2)$, so this is much worse than the $\mathcal{O}(n^2)$ performance of Dijkstra's shortest path algorithm, given in Figure 6.48, when all edge lengths are non-negative. For some sparse graphs, the algorithm could be faster than Dijkstra's. Of course, Dijkstra's algorithm cannot handle negative edges.

6.9.6.5 Edge Relaxation

The traditional implementation of the algorithm breaks the assignment

$$td[v] := \min(td[v], (\min u : u \in \text{pred}(v) : td[u] + e(u, v))) \quad (1)$$

into a number of simpler assignments, called *relaxing* the edge $u \rightarrow v$:

$$\text{for each edge } u \rightarrow v: td[v] := \min(td[v], td[u] + e(u, v)). \quad (2)$$

The computation in (2) is not as efficient as in (1) because it may assign values to $td[v]$ multiple times, once for each incoming edge, whereas in (1) $td[v]$ is

assigned only once. Both (1) and (2) require the same number of comparisons to compute the minimum. However, (2) is often preferred because it permits incremental computation in network protocols where edge lengths may change over time. The correctness of this implementation is explored in Exercise 21 (page 369).

6.10 Minimum Spanning Tree

A *spanning tree* of a connected undirected graph is a subset of its edges that forms a tree connecting all nodes of the graph. Given real-valued edge lengths, possibly negative, the length of a spanning tree is the sum of its edge lengths. A spanning tree of minimum length is called a *minimum spanning tree*, henceforth abbreviated as *mst*.

For the graph in Figure 6.51(a), $\{ab, bc, cd, de, ef, ej, hg, gi, ik, kl, lj\}$ with length 110 and $\{bd, ef, df, lj, gi, ac, bc, gk, kl, hg, ej\}$ with length 83 are both spanning trees. Clearly, the latter is a better choice based on its length. It is, in fact, the unique mst for this graph. Figure 6.51(b) shows this tree where its edges are in dashed lines. In general, a graph may have several mst. I will show that if the edge lengths are distinct, the mst is unique.

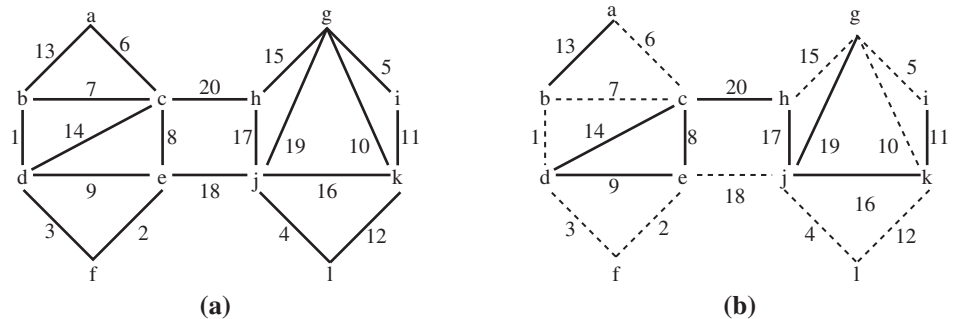


Figure 6.51 A graph with associated edge lengths and its minimum spanning tree.

Minimum spanning trees are essential in a number of applications. The most obvious is to build the shortest network to lay electrical or communication cables to link the nodes of a graph. It is used in operations research, cluster analysis, handwriting recognition and image processing. In short, it is a fundamental problem of graph theory that has applications in diverse areas.

There have been many algorithms for mst construction starting in the early 20th century. In this section, we study several well-known algorithms for this problem, due to Borůvka, [Kruskal \[1956\]](#), and an algorithm independently due to

Dijkstra [1959] and Prim [1957]. An optimal algorithm was invented by Pettie and Ramachandran [2002].

6.10.1 Properties of Spanning Tree

Henceforth, T denotes a spanning tree of a given graph of n nodes.

Lemma 6.4 ST1. T has $n - 1$ edges.

Proof. This is an elementary fact about trees; see a proof in Section 6.3.1. ■

It can be shown that any connected graph of n nodes that has $n - 1$ edges is a spanning tree.

ST2. There is a unique path connecting any pair of nodes in T .

Proof. There is a path between any two nodes x and y , from the definition of spanning tree. The path is unique because if there are two paths, p and p' , then $x \stackrel{p}{\rightsquigarrow} y \stackrel{p'}{\rightsquigarrow} x$ forms a cycle, which is impossible in a tree. ■

ST3. Adding an edge to T creates a cycle.

Proof. From property(ST2), $x \stackrel{p}{\rightsquigarrow} y$ exists in T and adding $x-y$ creates the cycle $x \stackrel{p}{\rightsquigarrow} y-x$. ■

Such a cycle is called a *fundamental cycle*. Each edge outside T creates a distinct fundamental cycle.

ST4. Removing *any* edge from a fundamental cycle yields a spanning tree.

Proof. Let $u-v$ be an edge in a fundamental cycle created by adding edge $x-y$ to T . I show that $T' = T - \{u-v\}$ is a spanning tree. Any two nodes, a and b , are connected in T' as follows. If the path $a \rightsquigarrow b$ in T does not include $u-v$, this path also exists in T' . Otherwise, a and b belong to the fundamental cycle in $T \cup \{x-y\}$ that includes $u-v$. Removing $u-v$ still leaves a and b connected. ■

Observe that T' has $n - 1$ edges (T has $n - 1$ edges, we added one edge and removed one to get T'), and all nodes are connected using the edges of T' ; so T' is a spanning tree.

6.10.2 A Fundamental Theorem About Minimum Spanning Trees

I give a condition for extending a partially constructed mst by adding new edges to it, in Theorem 6.13 (page 342). This theorem is the basis for all known algorithms that construct an mst by only adding edges.

6.10.2.1 Safe Edge

A given set of edges M partition the nodes of a graph into connected components, where two nodes are in the same component iff there is a path between them using only the edges of M . Edge $x-y$, where x and y belong to distinct components X and Y respectively, is an *inter-component* edge with respect to M ; the edges that are not inter-component are *intra-component*. Edge $x-y$ is *safe* for X wrt (with respect to) M if it is a shortest inter-component edge incident on X . Write $X \xrightarrow{e}_M Y$ to denote that edge e is incident on nodes in components X and Y , and e is safe for X under the partition induced by M ; I drop the subscript M when it is clear from the context. Note that e may not be safe for Y . A component may have several safe edges, and then they are of equal length, by definition. Call edge e safe wrt M if e is safe for *some* component X .

Example 6.2 Figure 6.52(a) shows the same graph as in Figure 6.51 in which every node is a component by itself, shown within a box, and the safe edges are shown as directed edges. Observe that edge $b-d$ is a safe edge for both components $\{b\}$ and $\{d\}$, so there is an arrow at each end of the edge, whereas $h-g$ is a safe edge for $\{h\}$ alone. Figure 6.52(b) shows two larger components, $\{a, c\}$ and $\{g, h, i\}$, and each remaining node as a component; the intra-component edges are shown as dashed lines. There is a new safe edge, $c-b$, between $\{a, c\}$ and $\{b\}$. The edge lengths are distinct in this example, so there is exactly one safe edge with an outgoing arrow for each component.

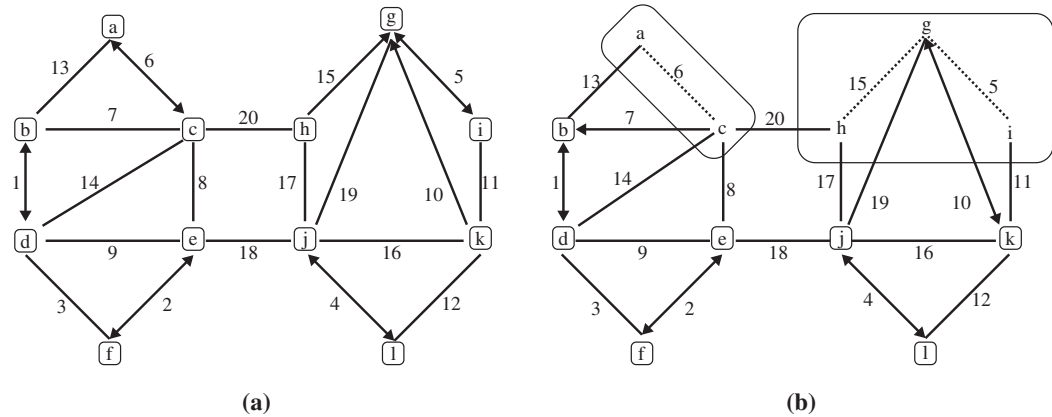


Figure 6.52 Different components of a graph.

6.10.2.2 Independence Among Safe Edges

The following lemma proves an independence result among safe edges.

Notation $\|e\|$ is the length of edge e and $\|M\|$ the sum of edge lengths in M . ■

Lemma 6.5 Let edges e and f of distinct lengths be safe wrt M . Then f is safe wrt $M \cup \{e\}$.

Proof. Suppose $X \xrightarrow{e}_M Y$ and f is safe for component Z . In $M \cup \{e\}$, components X and Y are merged to form a single component, call it XY . Consider the following cases for Z .

- $Z \notin \{X, Y\}$: The components wrt $M \cup \{e\}$ and M are the same except that X and Y are combined to form a component in $M \cup \{e\}$. Therefore, Z and its inter-component edges are unaffected by merging X and Y , so f is a safe edge for Z wrt $M \cup \{e\}$.
- $Z = X$: Both e and f are safe for X , so they have the same length, contradicting that e and f have distinct lengths.
- $Z = Y$: If f is incident on X , then both e and f are incident on both X and Y . Since e is safe for X , $\|e\| < \|f\|$, and f safe for Y implies $\|f\| < \|e\|$, contradiction.

Therefore, f is incident on Y and some components other than X . So, f is an inter-component edge of XY . I show that f is safe for component XY . Write $IE(X)$ and $IE(Y)$ for the inter-component edges of X and Y with respect to M , respectively, and $IE(XY)$ for the inter-component edges of XY wrt $M \cup \{e\}$. Then $IE(XY) \subseteq IE(X) \cup IE(Y)$.

$$\begin{aligned}
 & \text{true} \\
 \Rightarrow & \{e \text{ and } f \text{ are safe for } X \text{ and } Y, \text{ respectively}\} \\
 & \|e\| \leq (\min g : g \in IE(X) : \|g\|) \wedge \|f\| \leq (\min g : g \in IE(Y) : \|g\|) \\
 \Rightarrow & \{e \text{ is incident on } Y \text{ and } f \text{ is safe for } Y. \text{ So, } \|f\| < \|e\|\} \\
 & \|f\| \leq (\min g : g \in IE(X) : \|g\|) \wedge \|f\| \leq (\min g : g \in IE(Y) : \|g\|) \\
 \Rightarrow & \{IE(XY) \subseteq IE(X) \cup IE(Y)\} \\
 & \|f\| \leq (\min g : g \in IE(XY) : \|g\|) \\
 \Rightarrow & \{f \text{ is an inter-component edge of } XY\} \\
 & f \text{ is safe for } XY
 \end{aligned}$$

6.10.2.3 A Fundamental Theorem

Theorem 6.13 Let M be a subset of some mst and E a set of safe edges of distinct lengths. Then $M \cup E$ is a subset of some mst.

Proof. The proof is by induction on the size of E .

- E is empty: $M \cup E = M$ is a subset of some mst, by assumption.

• E has exactly one element e : Suppose M is a subset of mst M' . If $e \in M'$, then $M \cup \{e\} \subseteq M'$, so the proof is complete. Next, suppose $e \notin M'$ and e is a safe edge for component X . A property of a spanning tree is that $M' \cup \{e\}$ has a cycle. This cycle includes another edge f where $f \in M'$, and f is incident on X . Then $M'' = (M - \{f\}) \cup \{e\}$ is a spanning tree. I show that M'' is an mst because $\|M''\| \leq \|M'\|$.

Since e is a safe edge for X and f an inter-component edge of X , $\|e\| \leq \|f\|$. So, $\|M''\| = \|M'\| - \|f\| + \|e\| \leq \|M'\|$.

• E has more than one element: Let $e \in E$. Then $M \cup \{e\}$ is a subset of some mst, from the case proved above. From Lemma 6.5 (page 342), all edges of $E - \{e\}$ are safe wrt $M \cup \{e\}$. Inductively, $(M \cup \{e\}) \cup (E - \{e\}) = M \cup E$ is a subset of some mst. ■

A special case of Theorem 6.13 is of interest. A *cut* of a graph is a set of edges whose removal creates more than one connected component. It follows from Theorem 6.13 that an edge of minimum length in the cut belongs to some mst.

On distinct edge lengths for safe edges Theorem 6.13 requires that all edge lengths in E be distinct. To see the necessity of this condition, consider a graph of three nodes that are pairwise connected by edges of equal length. Given that each node is a component, each edge is safe. Inclusion of all the edges in E creates a cycle. The requirement of distinct edge lengths in E avoids this scenario.

The condition of distinct edge lengths is clearly met if E is a singleton. This is the property exploited in Kruskal's algorithm in Section 6.10.3 (page 345) and the Dijkstra-Prim algorithm in Section 6.10.4 (page 346). Borůvka's algorithm, Section 6.10.5 (page 348), is applied on graphs in which all edge lengths are distinct; so, the condition of the theorem is trivially met.

The requirement in Theorem 6.13 can be weakened; see Exercise 23 (page 369).

If all edges of the graph have distinct lengths, then the mst is unique (see Exercise 22, page 369). Theorem 6.13 has a simpler statement in this case: suppose M is a subset of the (unique) mst and E a set of safe edges; then $M \cup E$ is a subset of the mst.

6.10.2.4 Abstract Algorithm for MST

Theorem 6.13 suggests a procedure for constructing an mst. Start with an empty set for M , which is a subset any mst. Then every node is a component and all edges are inter-component. Identify the safe edges incident on each node, pick some subset E of these edges that have distinct lengths and add them to M . Repeat this step

until M is a spanning tree, then M is guaranteed to be a mst from Theorem 6.13. This procedure is described in the program in Figure 6.53.

Abstract algorithm for minimum spanning tree

```

 $M := \{\}$ ;
while  $M$  is not a spanning tree do
     $\{(\exists M' : M' \text{ is a mst} : M \subseteq M')\}$ 
     $E :=$  “a set of safe edges of distinct lengths wrt  $M$ ”;
     $M := M \cup E$ 
     $\{(\exists M' : M' \text{ is a mst} : M \subseteq M')\}$ 
enddo
 $\{(\exists M' : M' \text{ is a mst} : M \subseteq M'), M \text{ is a spanning tree}\}$ 
 $\{M \text{ is a mst}\}$ 

```

Figure 6.53

The abstract algorithm permits a number of choices for E . This allows for the development of a family of programs. Performance of different programs depend on the data structures and algorithms they employ for computing a set of safe edges of distinct lengths.

Example 6.3 Figure 6.54 shows two different choices of safe edges in the first step of applying Theorem 6.13 (page 342). In Figure 6.54(a), three edges are chosen from all available safe edges; the edges are shown as dashed lines and the resulting components are shown within boxes. In Figure 6.54(b), all safe edges are chosen; note that their lengths are all distinct.

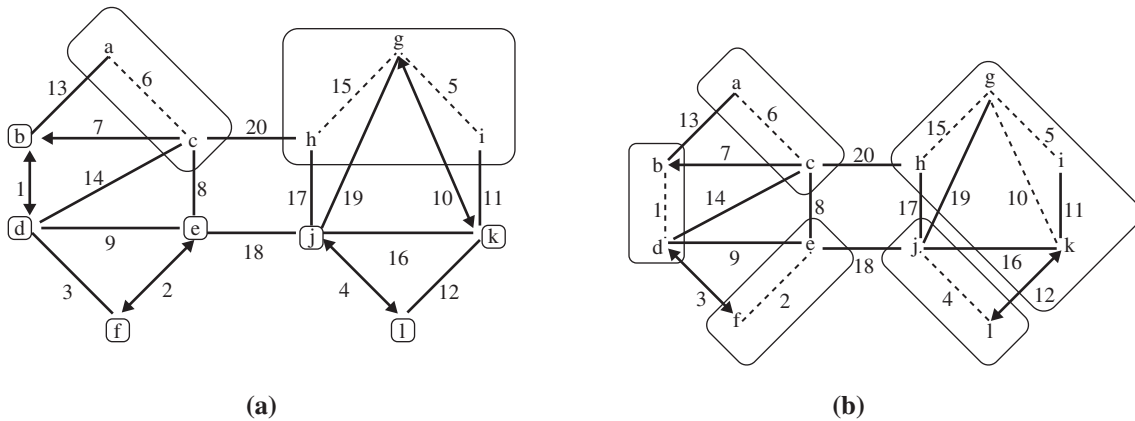


Figure 6.54 Different choices for mst edges.

6.10.3 Kruskal's Algorithm

The algorithm in Kruskal [1956] starts with an empty set for M . It scans the edges in order of increasing lengths where edges of equal length are scanned in arbitrary order. A scanned edge e is added to M if it is an inter-component edge, that is, $M \cup \{e\}$ has no cycle. Otherwise, it is discarded from further consideration. The steps continue until M is a spanning tree. This strategy meets the requirement of Theorem 6.13 (page 342) because any edge that is added is safe, being the shortest inter-component edge, and being a single edge meets the condition of distinct length.

Example 6.4 Figure 6.55 shows the execution of Kruskal's algorithm on the graph of Figure 6.51(a). The edges have distinct lengths from 1 through 20. So, the edge of length i is scanned in step i , $1 \leq i \leq 20$, unless the computation of the mst has been completed; in this case after edge $e - j$ of length 18 has been added. The edges that are chosen for inclusion in the mst are shown as dashed lines, the remaining ones as solid lines. Observe that $c - e$, $d - e$ and $i - k$ are discarded because each of them would have formed cycles with the chosen edges at that point, $b - d - f - e - c - b$ for $c - e$, $e - f - d - e$ for $d - e$, and $i - g - k - i$ for $i - k$.

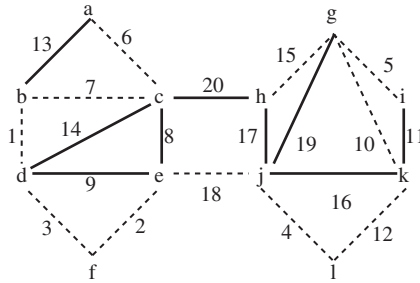


Figure 6.55 Selected edges during execution of Kruskal's algorithm.

Program for Kruskal's algorithm In the program of Figure 6.56, *unscanned* is the set of edges yet to be scanned, E is the set of edges of the graph, $|M|$ the size of set M and n the size of the graph. Assume that the graph is connected. The required invariant is: *unscanned* includes every inter-component edge. The correctness of the abstract algorithm for mst guarantees that this algorithm is correct (see Section 6.10.2.4, page 343).

Performance of the algorithm The algorithm performs two steps in each iteration: find a shortest edge e in *unscanned* and decide if e is an inter-component edge.

Kruskal's algorithm for minimum spanning tree

```

M, unscanned := {}, E;
{ (∃ M' : M' is a mst : M ⊆ M'), safeM ⊆ unscanned }
while |M| ≠ n − 1 do
    { (∃ M' : M' is a mst : M ⊂ M'), safeM ⊆ unscanned }
    e ∈ { f | f a shortest edge in unscanned };
    if M ∪ {e} has no cycle then M := M ∪ {e} else skip endif;
    unscanned := unscanned − {e}
    { (∃ M' : M' is a mst : M ⊆ M'), safeM ⊆ unscanned }
enddo
{ (∃ M' : M' is a mst : M ⊆ M'), |M| = n − 1 }
{ M is a mst }

```

Figure 6.56

Different data structures and algorithms have been proposed to implement each of these steps efficiently.

To decide if $x-y$ is an inter-component edge, that is, if $x-y$ does not belong to a cycle, use the Union-Find algorithm of Section 6.7.2 (page 305). Operation *find* determines if $x-y$ is inter-component, and *union* adds an edge to join two components. The Union-Find algorithm runs in near constant time per operation amortized over all operations. So, the running time of Kruskal's algorithm is dominated by the time it takes to find the next shortest edge.

A simple way to extract the shortest edges in order is to initially sort all edges, E , according to edge lengths. Then e is the next edge in this sorted list. The initial sorting step consumes $\mathcal{O}(m \log m)$ time, which is expensive for a dense graph (with $m = \mathcal{O}(n^2)$). Another possibility is to partition E into two subsets *low* and *high*, where each edge of *low* is shorter than each edge of *high*; use a proper variant of procedure *partition* from Section 5.4.3 (page 212). Then sort only *low* and scan its edges sequentially with the hope that the mst is a subset of *low*. If *low* does not include an mst, then partition *high* and scan the edges of its lower part in order. The goal is to avoid sorting a large set with the expectation that the mst is to be found among its shorter elements. Another common technique uses a priority queue to implement *unscanned*. Using a binary heap, the worst-case time of execution is $\mathcal{O}(\log m)$ to extract a shortest edge. The worst-case running time then is $\mathcal{O}(n \log m)$.

6.10.4 Dijkstra-Prim Algorithm

This algorithm was discovered independently by Dijkstra [1959] and Prim [1957]. At all points during the execution of the algorithm, there is a *primary* component (let

us call it *prim*; the coincidence in naming is entirely intended) that may consist of one or more nodes and remaining components that have exactly one node each. Initially, every node is a component by itself, and one of the components is chosen arbitrarily as the primary component. At all points, M is an mst over the nodes of the primary component. In each step a shortest inter-component edge incident on *prim* is added to M until M has $n - 1$ edges. The correctness of the algorithm is immediate from Theorem 6.13 (page 342).

Figure 6.57 shows the algorithm applied to the graph in Figure 6.51(a) where the edges of the mst are dashed lines. Initially, *prim* contains node g . Label $i : j$ on an edge denotes that the edge was chosen for inclusion in step i and its length is j . Observe that the order of inclusion of edges is dependent on the initial value of *prim*. In this example, many edges of higher length are included before the shortest edge $b-d$ of length 1 is included, in contrast with Kruskal's algorithm.

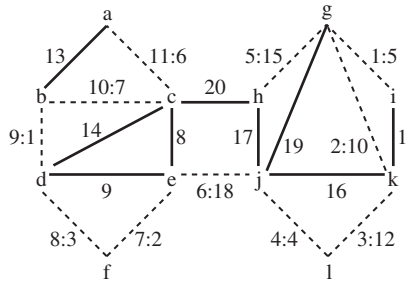


Figure 6.57 Edge choices in Dijkstra-Prim algorithm.

Next, I describe an efficient way of locating the required shortest edge. As before, $\|x-v\|$ is the length of edge $x-v$; take $\|x-v\|$ to be ∞ if the edge does not exist.

For every node x , $x \notin \text{prim}$, define $\text{cpn}[x]$ as the closest primary node to x , that is, the node in *prim* to which x has the shortest edge, and $\text{dp}[x]$ to be the length of this edge, that is, $\text{dp}[x] = \|x - \text{cpn}[x]\|$. Then the shortest inter-component edge incident on *prim* is $u-v$ where $u \notin \text{prim}$, $v \in \text{prim}$, $v = \text{cpn}[u]$, and $\text{dp}[u]$ the smallest value among all $\text{dp}[x]$. After adding $u-v$ to M and u to *prim*, the values $\text{cpn}[x]$ and $\text{dp}[x]$ have to be recomputed for all nodes x outside *prim* because their shortest edges to *prim* may be to node u . The computation, reminiscent of the similar computation in Dijkstra's shortest path algorithm, is as follows: if $\text{dp}[x] > \|x-u\|$ set $\text{cpn}[x]$ to u and $\text{dp}[x]$ to $\|x-u\|$. The complete algorithm appears in Figure 6.58.

Performance of the algorithm The algorithm executes $n-1$ iterations. In each iteration, it computes the minimum over a set of size at most n and updates at most two values for at most n nodes. Therefore, the algorithm executes $\mathcal{O}(n^2)$ steps. This

Dijkstra-Prim algorithm for minimum spanning tree

```

{  $V$  is the set of all nodes }
 $M, prim := \{\}, \{a\}$ , where  $a$  is an arbitrary node;
for  $x \in V - \{a\}$  do  $cpn[x], dp[x] := a, \|x - a\|$  endfor;
while  $|M| \neq n - 1$  do
   $u := \{y \mid y \notin prim, dp[y] = (\min x : x \notin prim : dp[x])\}$ ;
   $v := cpn[u]$ ;
   $M, prim := M \cup \{u - v\}, prim \cup \{u\}$ ;
  for  $x \in V - prim$  do
    if  $dp[x] > \|x - u\|$ 
      then  $cpn[x], dp[x] := u, \|x - u\|$ 
    else skip
    endif
  endfor
enddo

```

Figure 6.58

is preferable to Kruskal's algorithm for dense graphs. For sparse graphs, Kruskal's algorithm is usually superior.

6.10.5 Borůvka's Algorithm

Perhaps the first mst algorithm was designed by Borůvka in 1926. The algorithm has been discovered and rediscovered several times. Typically, the algorithm is applied on graphs in which all edge lengths are distinct. Initially $M = \{\}$. In each step let E be the set of all safe edges wrt M . This set is unique because edge lengths are distinct. It is a greedy algorithm in the sense that it does all that is possible in a step. Figure 6.54(b) shows the first step of the algorithm applied to the graph in Figure 6.51(a). The next step will choose $\{b-c, d-f, j-k\}$ as E and the final step $\{e-j\}$.

Every step reduces the number of components by at least half, usually more, so the number of steps is bounded by $\log_2 n$. Each step, of course, takes longer than joining just two components.

The algorithm permits efficient parallel implementation. Large graphs can be processed by multiple computers simultaneously because it is sufficient to find a safe edge of a component independent of other components. This parallel version of the algorithm was popularized by Sollin [1965].

6.11 Maximum Flow

Maximum flow is a problem of considerable importance in operations research. It models the situation where some commodity is to be transferred in a graph from a source node to a sink node via a number of intermediate nodes that can only transfer, but not store, the commodity. Each edge has a capacity constraint that limits the amount of commodity that can be transferred over it. The goal is to find a schedule, called a *flow*, by which the maximum possible amount of the commodity can be transferred from the source to the sink. The problem arises in transferring goods over a railway network, the original inspiration for the problem, and also in a variety of other problems including data communication networks. Many combinatorial problems, for example, determining the maximum number of edge-disjoint paths between a pair of given nodes in a directed graph or computing a maximum matching in a bipartite graph, can be cast as maximum flow problems.

The problem was first formulated and solved by [Ford and Fulkerson \[1962\]](#) in their classic book *Flows in Networks*. They proved a fundamental theorem, known as the max-flow min-cut theorem. Based on this theorem, they gave an algorithm for computing the maximum flow, which has a running time proportional to the number of edges and the value of the maximum flow. Since the publication of their book, there have been a vast number of papers on finding more efficient algorithms for this problem. I discuss an algorithm by [Goldberg and Tarjan \[1988\]](#) in Section 6.12 (page 356) that runs in $\mathcal{O}(n^2 \cdot m)$ time where n is the number of nodes and m the number of edges. Algorithms by [King et al. \[1994\]](#) and [Orlin \[2013\]](#) establish a $\mathcal{O}(n \cdot m)$ time bound.

6.11.1 Problem Specification

A *network* is a graph in which the set of nodes, V , includes two special nodes, s for *source* and t for *sink*; nodes other than s and t are *intermediate* nodes. Assume that if edge $u \rightarrow v$ exists, so does $v \rightarrow u$. Every edge $u \rightarrow v$ has an associated non-negative integer $c(u, v)$, called its *capacity*.

A *flow* f satisfies the constraints (F0, F1), below. A *flow* f in network with capacities c additionally satisfies the constraint (F2).

F0. (antisymmetry) $f(u, v) = -f(v, u)$, for all edges $u \rightarrow v$.

F1. (flow conservation) $(+v : v \in V : f(v, u)) = 0$, for $u \notin \{s, t\}$.

F2. (capacity constraint) $f(u, v) \leq c(u, v)$, for all edges $u \rightarrow v$.

Constraint (F0) says that a flow along an edge induces a flow of the opposite magnitude in the reverse direction, (F1) that the total incoming flow to an intermediate node is 0, and (F2) that the flow along an edge cannot exceed its capacity. It is easy to show from these constraints that the outflow at any intermediate node is also 0, just like the inflow.

Notation Write $f_{in}(v)$ and $f_{out}(v)$, respectively, for *inflow* and *outflow* for any node v ; they are $(+u : u \in V : f(u, v))$ and $(+u : u \in V : f(v, u))$, respectively. An edge with positive capacity is a *positive edge*, and a simple path in which all edges are positive is a *positive path*. A positive path from s to t is also called a *flow augmenting path* in the literature.

Flow value, maximum flow The *value* of flow f , written as $|f|$, is $f_{out}(s)$, the outflow from s . It can be shown that $|f| = f_{in}(t)$, the inflow to t . The flow value may be negative, zero or positive. The max flow problem is to compute a flow having the maximum value. Maximum flow value is positive under mild restrictions on capacities and connectivity in the network, as shown in Observation 6.5.

Observation 6.5 It is possible to send a positive flow from s to t iff there is a positive path from s to t .

Proof.

1. If there is a positive $s \rightsquigarrow t$ path, a unit flow (a flow of unit value) can be sent along it from s to t .
2. If it is possible to send a positive flow from s to t , it is possible to send a unit flow. The unit flow follows a positive $s \rightsquigarrow t$ path. ■

Example 6.5 Figure 6.59(a) shows a network with capacities on its edges. Capacities are zero for the edges that are not shown. In Figure 6.59(b), each edge is labeled with a flow; the flows of zero or negative values are not shown. Verify that this flow satisfies the conditions (F0, F1, F2) with the capacities shown in Figure 6.59(a). The flow value, the outflow from s , is 18.

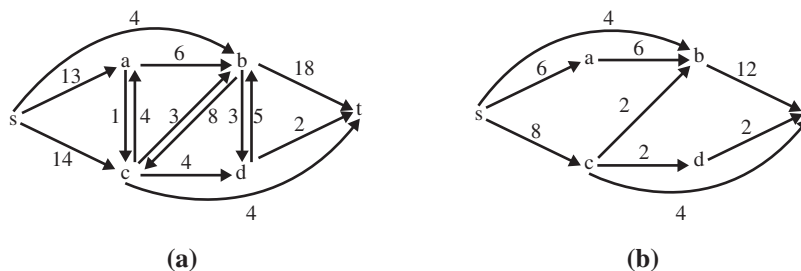


Figure 6.59 A capacitated network and a flow in it.

Note In the original work of Ford and Fulkerson [1962], flow is defined slightly differently. The capacity constraint, (F2), is modified to $0 \leq f(u, v) \leq c(u, v)$, and the antisymmetry constraint says that at least one of $f(u, v)$ and $f(v, u)$ is 0. This is a more intuitively appealing description of a flow, that the flow along an edge is non-negative and bounded by the edge capacity, and that the flow goes from u to v or from v to u , but not in both directions. The flow conservation constraint is that the total incoming and outgoing flows at an intermediate node are equal.

I develop the max-flow theory with constraints (F0), (F1) and (F2) because the algebra of flows is simpler in this model. In particular, it is possible to add and subtract two flows edge by edge, which is not possible with Ford–Fulkerson flows. We can convert flow f to a Ford–Fulkerson flow f' by $f'(u, v) = \max(0, f(u, v))$ for every edge $u \rightarrow v$, that is, simply converting a negative flow to 0. Conversely, f' can be converted to a flow f by $f(u, v) = f'(u, v) - f'(v, u)$. ■

6.11.2 Residual Network

Given flow f in c , define $c_f(u, v)$ to be $c(u, v) - f(u, v)$ for every edge $u \rightarrow v$, regardless of the value of $f(u, v)$, positive, zero or negative. Call $c_f(u, v)$ the *residual capacity* of edge $u \rightarrow v$ and the network with capacities c_f the *residual network*, with respect to f . Since f satisfies constraint (F2), $f(u, v) \leq c(u, v)$, so $c_f(u, v)$ is non-negative.

Example 6.6 Figure 6.60(a) shows the pair (flow, capacity) on each edge, using the capacities and flow from Figure 6.59 (page 350); for an edge along which there is no flow, only the capacity is shown. The corresponding residual network is shown in Figure 6.60(b). I show a positive path in this figure using dashed edges. Since the minimum capacity of any edge along the path, of edge (c, d) , is 2, a flow of 2 units can be sent along this path from s to t .

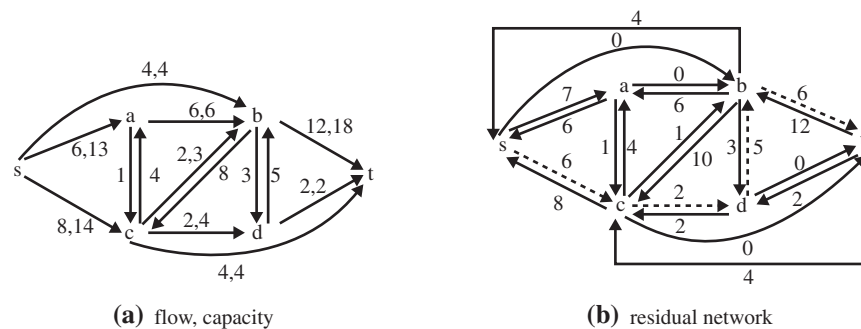


Figure 6.60 Residual network; a positive path is dashed.

6.11.3 Computing a Maximum Flow

For flows f and g , define their sum $f + g$ and difference $f - g$ by:

$$\begin{aligned}(f + g)(u, v) &= f(u, v) + g(u, v), \text{ and} \\ (f - g)(u, v) &= f(u, v) - g(u, v), \text{ for all } u \rightarrow v.\end{aligned}$$

Since f and g are flows, they satisfy (F0, F1), and so do $f + g$ and $f - g$ because these constraints are independent of the capacity. However, for flows f and g in c , $f + g$ and $f - g$ may not satisfy (F2). It is easily seen that if $f + g$ and $f - g$ are flows, then $|f + g| = |f| + |g|$ and $|f - g| = |f| - |g|$. The sum and difference of flows obey the standard algebraic laws of addition and subtraction, so $(f + g) - f = g$, for example.

Observation 6.6 Given flow f in c , $f + g$ is a flow in c iff g is a flow in the residual network c_f .

Proof.

$$\begin{aligned}& f + g \text{ is a flow in } c \\ \equiv & \{ \text{definition of flow in } c \} \\ & f + g \text{ is a flow, and (F2): } (\forall u \rightarrow v :: (f + g)(u, v) \leq c(u, v)) \\ \equiv & \{ f + g \text{ is a flow and } f \text{ is a flow, so } g = (f + g) - f \text{ is a flow,} \\ & (f + g)(u, v) \leq c(u, v) \equiv g(u, v) \leq c(u, v) - f(u, v) \} \\ & g \text{ is a flow, } (\forall u \rightarrow v :: g(u, v) \leq c(u, v) - f(u, v)) \\ \equiv & \{ \text{definition of } c_f \} \\ & g \text{ is a flow in } c_f\end{aligned}$$

Corollary 6.2 Let f be a flow in c . Then $f + g$ is a maximum flow in c iff g is a maximum flow in c_f .

Proof. Given flow f in c , there is a 1-1 correspondence between the flows in c and c_f , from Observation 6.6. Further, for the corresponding flows $f + g$ and g in c and c_f , respectively, $|f + g| = |f| + |g|$. Therefore, $f + g$ is a maximum flow in c iff g is a maximum flow in c_f . ■

Observations 6.5 (page 350) and 6.6 (page 352) suggest a computation method for maximum flow. As long as there is a positive path in the given network, send as much flow as possible along it (which is the minimum capacity over the edges in that path) and modify the network to the residual network corresponding to this flow. The program in Figure 6.61 uses the invariant

$$\text{Inv}:: (f + \text{the max flow in } c_f) = \text{the max flow in } c.$$

Initially, the flow along all edges is 0, so the residual capacities c_f are the same as the initial capacities c . Finally, there is no positive $s \rightsquigarrow t$ path in the residual network, so from Observation 6.5, the computed flow is maximum. Termination is guaranteed because each iteration increases the flow value $|f|$, and the maximum flow value is bounded.

Ford-Fulkerson max flow algorithm

```

 $c_f := c;$ 
{ initial flow along all edges,  $E$ , is 0 }
for  $u \rightarrow v \in E$  do  $f(u, v) := 0$  endfor;
{  $Inv$  }
while there is a positive path  $s \overset{p}{\rightsquigarrow} t$  in the residual network  $c_f$  do
   $minc :=$  minimum capacity over edges in  $p$ ;
  for  $(u, v) \in \{(i, j) \mid i \rightarrow j \in p\}$  do
     $f(u, v), c_f(u, v) := f(u, v) + minc, c_f(u, v) - minc;$ 
     $f(v, u), c_f(v, u) := f(v, u) - minc, c_f(v, u) + minc;$ 
  endfor
enddo
{  $f$  is a maximum flow in  $c$  }

```

Figure 6.61

This was the first algorithm designed for this problem, described (using different notation) in [Ford and Fulkerson \[1962\]](#). Each iteration requires finding a positive path, which takes $\mathcal{O}(m)$ steps where m is the number of edges. The number of iterations may be as high as M , the value of the maximum flow, because just one unit of flow may be added in each iteration. So, the algorithm has a running time of $\mathcal{O}(m \cdot M)$. Current algorithms run in polynomial time in the size of the network independent of the value of the maximum flow.

Example 6.7 Consider the capacitated network in Figure 6.59(a). Figure 6.62(a) shows (flow, capacity) along each edge of that network. The corresponding residual network is shown Figure 6.62(b). The given flow is maximum because there is no positive path in the residual network. The flow value, the outgoing flow from s , is 21.

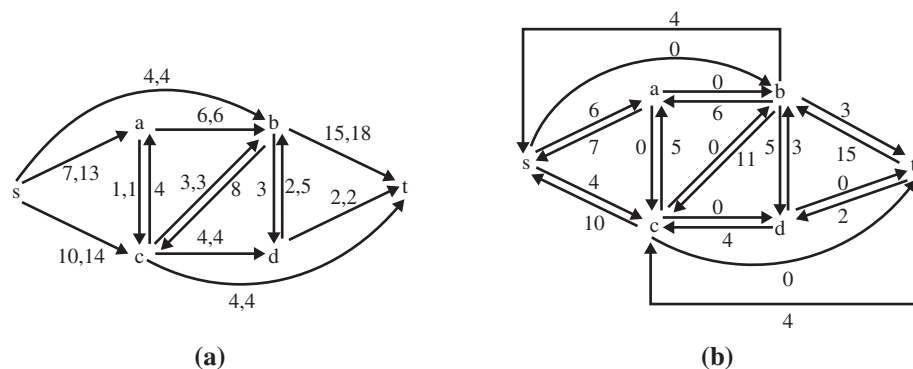


Figure 6.62 Residual network for a maximum flow.

6.11.4 Max-flow Min-Cut Theorem

6.11.4.1 Algebra of Flows

For flow f and sets of nodes X and Y , write $f(X, Y)$ for the total flow from nodes in X to nodes in Y , that is,

$$f(X, Y) = (+u, v : u \in X, v \in Y : f(u, v)).$$

Then $f_{out}(X) = f(X, V)$ and $f_{in}(X) = f(V, X)$. Using the convention to write x for a singleton set $\{x\}$, $f_{out}(x) = f(x, V)$ and $f_{in}(x) = f(V, x)$. Adopt the convention that $f(x, x) = 0$.

In flow networks, a *cut* is a partition of the vertices into parts S and T such that $s \in S$ and $t \in T$. The set of edges $u \rightarrow v$, where $u \in S$ and $v \in T$ is called a *cut-set*; I will simply call it a cut and denote it by (S, T) .

Lemma 6.6 For flow f and sets of nodes X , Y and Z :

1. $f(X, X) = 0$.
2. (Distributivity) For disjoint sets Y and Z :

$$f(X, Y \cup Z) = f(X, Y) + f(X, Z),$$

$$f(Y \cup Z, X) = f(Y, X) + f(Z, X).$$

Similar distributivity rule applies for f_{out} and f_{in} .

3. For any cut (S, T) , $f_{out}(S) = f_{out}(s) = f_{in}(t) = f_{in}(T)$.
In particular, $|f| = f(S, T)$. ■

It is easy to prove this lemma; see Exercise 26 (page 370) for the proof of part (3). As a simple application of the lemma, I derive:

Observation 6.7 $f_{in}(X) = f(V - X, X)$, $f_{out}(X) = f(X, V - X)$.

Proof.

$$\begin{aligned} & f_{in}(X) \\ = & \{f_{in}(X) = f(V, X) \text{ Apply distributivity on } f(V, X)\} \\ & f(X, X) + f(V - X, X) \\ = & \{\text{from part (1) of the lemma } f(X, X) = 0\} \\ & f(V - X, X) \end{aligned}$$

Prove $f_{out}(X) = f(X, V - X)$ similarly. ■

6.11.4.2 Statement and Proof of Max-Flow Min-Cut Theorem

The capacity of cut (S, T) is the sum of the capacities of all the edges from S to T , that is, $(+u, v : u \in S, v \in T : c(u, v))$. Extending the notation for flows to capacities, this is $c(S, T)$. A *minimum cut* has the minimum capacity over all cuts.

Theorem 6.14 max-flow min-cut

Let f be a maximum flow and (S, T) a minimum cut. Then $|f| = c(S, T)$.

Proof.

• (1) $|f| \leq c(S, T)$: I prove this result for *any* flow f and *any* cut (S, T) ; so it holds for the special case of maximum flow and minimum cut.

$$\begin{aligned}
 & |f| \\
 = & \quad \{\text{from Lemma 6.6, } |f| = f(S, T)\} \\
 & \quad (+u, v : u \in S, v \in T : f(u, v)) \\
 \leq & \quad \{\text{from constraint (F2), } f(u, v) \leq c(u, v)\} \\
 & \quad (+u, v : u \in S, v \in T : c(u, v)) \\
 = & \quad \{\text{definition of } c(S, T)\} \\
 & \quad c(S, T)
 \end{aligned}$$

• (2) For a maximum flow f there is a cut (S, T) such that $|f| = c(S, T)$:

Since f is a maximum flow in c , the maximum flow in c_f has value 0, from Corollary 6.2 (page 352). Let $S = \{s\} \cup \{x \mid \text{there is a positive path from } s \text{ to } x \text{ in } c_f\}$. From Observation 6.5 (page 350), there is no positive path from s to t in c_f , so $t \notin S$. Let $T = V - S$, so, (S, T) is a cut.

For any edge (u, v) where $u \in S$ and $v \in T$:

$$\begin{aligned}
 & \text{true} \\
 \Rightarrow & \quad \{c_f(u, v) = 0, \text{ otherwise there is a positive path } s \rightsquigarrow u \rightarrow v \\
 & \quad \text{in the residual network}\} \\
 & \quad c_f(u, v) = 0 \\
 \Rightarrow & \quad \{(+u, v : u \in S, v \in T : c_f(u, v)) = 0\} \\
 & \quad c_f(S, T) = 0 \\
 \Rightarrow & \quad \{c_f(S, T) = c(S, T) - f(S, T). \text{ From Lemma 6.6 (page 354) } |f| = f(S, T)\} \\
 & \quad |f| = c(S, T)
 \end{aligned}$$

From (1, 2) the maximum flow value equals the minimum cut capacity. ■

Example 6.8 I show a minimum cut for the network of Figure 6.59(a) (page 350). Take the residual network corresponding to the max flow, Figure 6.62(b) (page 353), which I reproduce in Figure 6.63(a). Obtain a cut of the network of Figure 6.59(a) as shown in Figure 6.63(b), applying the technique described in Theorem 6.14, where the nodes to the left of the vertical dashed line are in S and to the right in T . Verify that the corresponding cut capacity is $4 + 6 + 3 + 4 + 4 = 21$, same as the value of the max flow.

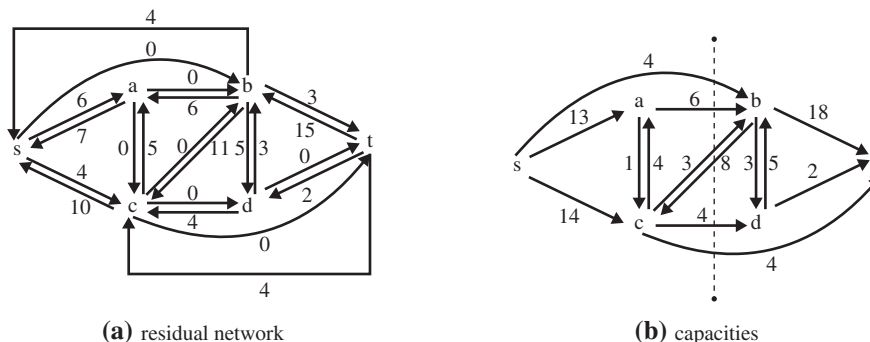


Figure 6.63 A minimum cut of a capacitated network.

6.12 Goldberg–Tarjan Algorithm for Maximum Flow

The Ford–Fulkerson algorithm for maximum flow (Figure 6.61, page 353) has a worst-case running time of $\mathcal{O}(m \cdot M)$, where m is the number of edges and M the magnitude of the maximum flow. It is desirable to design an algorithm whose running time is a polynomial in the parameters of the network, the number of nodes and edges, independent of the flow value. Goldberg and Tarjan [1988] propose an algorithm with running time of $\mathcal{O}(m \cdot n^2)$, where m is the number of edges and n the number of nodes.

The Ford–Fulkerson algorithm obeys the constraints (F0, F1, F2) of Section 6.11.1 (page 349) at all times *during* the execution of the algorithm. The Goldberg–Tarjan algorithm obeys only (F0, F2) during its execution and a constraint weaker than the flow conservation constraint (F1): that $f_{in}(u) \geq 0$ for all u , $u \neq s$. Such a flow is called a *preflow*. At termination of the algorithm, the preflow obeys (F1), so it is a flow.

Outline of the algorithm The algorithm starts by sending a preflow of maximum possible value along all outgoing edges of s . This step creates certain nodes that have more inflow than outflow, that is, $f_{in}(u) > 0$ for such nodes. Call an *intermediate* node u *active* if $f_{in}(u) > 0$. The goal of the algorithm is to disperse extra inflow at active nodes so that preflow becomes a flow. The algorithm uses two basic operations, *push* and *relabel*, for this purpose.

A *push* operation is applied at an active node u that has a positive outgoing edge $u \rightarrow v$ with residual capacity $c_f(u, v)$. Maximum possible flow is sent along this edge, which is $\min(f_{in}(u), c_f(u, v))$. However, repeated application of this strategy will merely circulate flow along a cycle. So, an additional condition is imposed. Each node is assigned an integer-valued *height*, $h(u)$ for node u , and a flow is sent from u to v only if $h(u) > h(v)$. That is, flow can go only from higher to lower height.

The heights of the intermediate nodes have to be modified during the course of the algorithm, otherwise the graph will have a fixed acyclic structure based on node heights and excess flow can never be sent back to any node. This is done through a *relabel* operation: if active node u has equal or lower height than all its successors, then its height is increased to the minimum possible value so that it is higher than some successor.

The algorithm, after initialization, repeatedly chooses an arbitrary active node. It applies push or relabel, whichever is applicable, at that node; I will show that exactly one of these operations is applicable at any active node. The algorithm terminates when there is no active node. I show that the preflow is then a flow, and it is the maximum flow.

6.12.1 Details of the Algorithm

Convention The value of a preflow (hence, a flow) can be computed from the initial capacities c and the residual capacities c_f at any point in the algorithm, using the invariant $f(u, v) = c(u, v) - c_f(u, v)$, for any $u \rightarrow v$. Therefore, the program does not explicitly compute the preflows during the execution of the algorithm, instead it computes only the residual capacities.

6.12.1.1 Initialization

Initially, the residual capacities c_f are the same as c . The height of s is set to n , the number of nodes, and for other nodes to 0. Maximum possible flow is sent along all outgoing edges of s , and the capacities recomputed for the residual network. Below, V is the set of all nodes and E the set of all edges.

```

initialize::
   $c_f = c$ ;
   $h(s) := n$ ;
  for  $u \in V - \{s\}$  do  $h(u) := 0$  endfor;
  for  $u \in \{v \mid s \rightarrow v \in E\}$  do  $c_f(u, s) := c(u, s) + c(s, u)$ ;  $c_f(s, u) := 0$  endfor

```

6.12.1.2 Push Operation

An active node u sends the maximum possible flow along positive edge $u \rightarrow v$ to successor v of lower height.

```

push( $u, v$ )::
   $\{f_{in}(u) > 0, c_f(u, v) > 0, h(u) > h(v)\}$ 
   $d := \min(f_{in}(u), c_f(u, v))$ ;
   $c_f(u, v), c_f(v, u) := c_f(u, v) - d, c_f(v, u) + d$ 

```

6.12.1.3 Relabel Operation

I will show later that if an active node u cannot do a push, because it has no edge $u \rightarrow v$ where $c_f(u, v) > 0$ and v has lower height, then it can do a relabel operation, that is, the set $\{v \mid c_f(u, v) > 0\}$ is then non-empty.

```

relabel( $u$ )::
 $\{f_{in}(u) > 0, (\forall v : c_f(u, v) > 0 : h(u) \leq h(v))\}$ 
 $h(u) := 1 + (\min v : c_f(u, v) > 0 : h(v))$ 

```

6.12.1.4 The Complete Algorithm

The program in Figure 6.64 (page 358) uses two derived variables:

- (1) preflow along each edge $u \rightarrow v$ is given by $f(u, v) = c(u, v) - c_f(u, v)$, and
- (2) the set of active nodes by $active = \{u \mid u \notin \{s, t\}, f_{in}(u) > 0\}$.

The annotation in Figure 6.64 uses invariant I , which is defined in Section 6.12.2.1.

Goldberg-Tarjan max flow algorithm

```

initialize;
{  $I$  }
while  $active \neq \{\}$  do
   $u \in active$ ;
  {  $I, f_{in}(u) > 0$  }
  { compute the set of successors of  $u$  such that  $c_f(u, v) > 0 \wedge h(u) > h(v)$  }
   $succu := \{v \mid c_f(u, v) > 0 \wedge h(u) > h(v)\}$ ;
  if  $succu \neq \{\}$ 
    then  $v \in succu$ ;
    {  $I, f_{in}(u) > 0, c_f(u, v) > 0 \wedge h(u) > h(v)$  }
     $push(u, v)$ 
    {  $I$  }
  else
    {  $I, f_{in}(u) > 0, (\forall v : c_f(u, v) > 0 : h(u) \leq h(v))$  }
     $relabel(u)$ 
    {  $I$  }
  endif
{  $I$  }
enddo ;
{  $I, active = \{\}$  }
{ The preflow is a maximum flow }

```

Figure 6.64

6.12.2 Correctness

The partial correctness of the program of Figure 6.64 involves showing the validity of the annotation in that figure. Invariant I is postulated and proved in Section 6.12.2.1. The remaining part of the annotation, that the preconditions of push and relabel hold, is proved in Section 6.12.2.2. The postcondition of the loop, that the preflow is a maximum flow, is proved in Section 6.12.2.3 (page 361). Proof of termination is given in Section 6.12.2.4 (page 361).

6.12.2.1 Invariant

Invariant I , used in the annotation in Figure 6.64 (page 358), is the conjunction of four parts, (I0, I1, I2, I3):

- (I0). f is a preflow, that is,
 - F0. (antisymmetry) $f(u, v) = -f(v, u)$, for all edges $u \rightarrow v$.
 - F1'. (inflow) $f_{in}(u) \geq 0$, for $u \neq s$.
 - F2. (capacity constraint) $f(u, v) \leq c(u, v)$, for all edges $u \rightarrow v$.
- (I1). Every active node has a positive path to s in the residual network.
- (I2). For a positive path $u \rightsquigarrow w$ with e edges $h(u) \leq e + h(w)$.
- (I3). Height of each node is non-decreasing, and
 - $h(s) = n$, $h(t) = 0$, and $(\forall u :: 0 \leq h(u) < 2 \cdot n)$.

Invariant I is proved in the standard fashion: show (1) I holds following *initialize*, and (2) I is preserved by the executions of *push* and *relabel*. Proof of (I0) is by inspection in all cases, which I omit. I show the remaining proofs below.

- Proof of (I1): This result follows from (I0). Let u be an active node and R the set of nodes that includes u and the nodes reachable from u via a positive path.

First, I show that $f_{in}(R) \leq 0$. For any $x \in R$, $y \in V - R$:

$$\begin{aligned}
 & x \in R, y \in V - R \\
 \Rightarrow & \{ \text{there is a positive path } u \rightsquigarrow x, \text{ but no positive path } u \rightsquigarrow y \text{ because } y \notin R \} \\
 & c_f(x, y) = 0 \\
 \Rightarrow & \{ f(x, y) = c(x, y) - c_f(x, y), \text{ and } c(x, y) \geq 0 \} \\
 & f(x, y) \geq 0 \\
 \Rightarrow & \{ \text{antisymmetry} \} \\
 & f(y, x) \leq 0 \\
 \Rightarrow & \{ f(V - R, R) = (+x, y : x \in R, y \in V - R : f(y, x)) \} \\
 & f(V - R, R) \leq 0 \\
 \Rightarrow & \{ \text{from Observation 6.7 (page 354)} f_{in}(R) = f(V - R, R) \} \\
 & f_{in}(R) \leq 0
 \end{aligned}$$

Next, I show $s \in R$. Let $R' = R - \{s\}$.

$$\begin{aligned}
 & \text{true} \\
 \Rightarrow & \{ \text{from (F1')} f_{in}(x) \geq 0, \text{ for } x \neq s. \text{ And } s \notin R'. \\
 & u \text{ is active and } u \in R', \text{ so } f_{in}(u) > 0. \} \\
 & f_{in}(R') > 0 \\
 \Rightarrow & \{ \text{from the last proof, } f_{in}(R) \leq 0 \} \\
 & R \neq R' \\
 \Rightarrow & \{ R' = R - \{s\} \} \\
 & s \in R
 \end{aligned}$$

Initialization establishes (I2) and (I3)

• Proof of (I2):

The active nodes that have a positive path to s are of the form $u \rightarrow s$. Since $h(u) = 0$ and $h(s) = n$, $h(u) \leq 1 + h(s)$.

• Proof of (I3): by inspection.

The push operation preserves (I2) and (I3)

• Proof of (I2):

I only prove that for every positive edge $x \rightarrow y$, $h(x) \leq 1 + h(y)$. Then (I2), that for a positive path $u \rightsquigarrow w$ with e edges $h(u) \leq e + h(w)$, follows by induction on e .

Suppose $h(x) \leq 1 + h(y)$ holds for all positive edges $x \rightarrow y$ before a $push(u, v)$. Following the push, some of the previously positive edges may not be positive, which does not affect this assertion. The only positive edge that may be created is $v \rightarrow u$. From the precondition of $push(u, v)$, $h(u) > h(v)$ holds, and since the heights are not modified by push, $h(v) \leq 1 + h(u)$ holds afterwards.

• Proof of (I3):

The push operation does not modify heights, so (I3) is preserved.

The relabel operation preserves (I2) and (I3)

• Proof of (I2):

I show that following a relabel of u if $u \rightarrow v$ is a positive edge, $h(u) \leq 1 + h(v)$. Then (I2) follows by induction, as in the proof for the $push$ operation.

It is easy to justify the following annotation.

$$\begin{aligned}
 & \{ (\forall v : c_f(u, v) > 0 : h(u) \leq h(v)) \} \\
 & \{ h(u) \leq (\min v : c_f(u, v) > 0 : h(v)) \} \\
 & \quad h(u) := 1 + (\min v : c_f(u, v) > 0 : h(v)) \\
 & \{ h(u) \leq 1 + (\min v : c_f(u, v) > 0 : h(v)) \} \\
 & \{ (\forall v : c_f(u, v) > 0 : h(u) \leq 1 + h(v)) \}
 \end{aligned}$$

• Proof of (I3): The relabel operation does not modify the height of s or t , so they continue to have their initial values. For any intermediate node u , a $\text{relabel}(u)$ only increases $h(u)$. Therefore, the heights are non-negative and non-decreasing.

To prove that $h(u) < 2 \cdot n$, a $\text{relabel}(u)$ modifies $h(u)$ only if u is active before the operation. It remains active after the operation. From (I1) u has a positive path to s after the operation, say of length e .

$$\begin{aligned}
 & h(u) \\
 \leq & \quad \{\text{from (I2)}\} \\
 & e + h(s) \\
 \leq & \quad \{e \text{ is at most } n - 1; h(s) = n\} \\
 & (n - 1) + n \\
 < & \quad \{\text{arithmetic}\} \\
 & 2 \cdot n
 \end{aligned}$$

6.12.2.2 Preconditions of push and relabel

I show that the preconditions of push and relabel shown in the annotation in Figure 6.64 (page 358) hold.

For $\text{push}(u, v)$, I holds by assumption. And the choice of u and v guarantees

$$f_{in}(u) > 0, c_f(u, v) > 0, h(u) > h(v).$$

The precondition of relabel holds before its execution by the proof rule for **if**. However, we have to show that for active u the set $\{v \mid c_f(u, v) > 0\}$ is never empty. This follows from (I1): since u is active, it has a positive path to s , so $c_f(u, v) > 0$ for some v .

6.12.2.3 The Preflow is Max Flow at Termination

At termination, there is no active node. So, $f_{in}(u) = 0$ for every intermediate node, that is, the preflow is a flow. To prove that the flow is then a maximum flow, I show that there is no positive $s \rightsquigarrow t$ path. If there is such a path of length e , from (I2) $h(s) \leq e + h(t)$, and from (I3) $h(t) = 0$ and $h(s) = n$. So $n \leq e$, a contradiction because any (simple) path length is at most $n - 1$.

6.12.2.4 Termination

I prove a strong time bound for this algorithm in Section 6.12.3, which is sufficient for a proof of termination. Here, I use a simpler argument to prove termination, though it does not give a sharp bound on the complexity.

Let ch be the cumulative heights of the intermediate nodes. I show that the following well-founded metric decreases lexicographically by the execution of push and relabel; so it decreases in each iteration:

$$((n - 2) \times (2 \cdot n - 1) - ch, (+u : u \text{ an intermediate node} : f_{in}(u) \times h(u))).$$

From (I3), each intermediate node's height is at least 0 and at most $2 \cdot n - 1$. Then, for $n - 2$ intermediate nodes, $0 \leq ch \leq (n - 2) \times (2 \cdot n - 1)$, so the first term is non-negative. Execution of *relabel*(u) increases the height of u ; so, it increases ch and decreases the first component of the metric, thus decreasing the metric lexicographically.

Execution of *push*(x, y) does not modify heights, so $(n - 2) \times (2 \cdot n - 1) - ch$ remains same. It moves a positive amount of flow from x to y where $h(x) > h(y)$. Therefore, it decreases $f_{in}(x) \times h(x) + f_{in}(y) \times h(y)$; so the metric $(+u : u \text{ an intermediate node} : f_{in}(u) \times h(u))$ decreases.

6.12.3 Timing Analysis

I show that the number of *relabel* operations is $\mathcal{O}(n^2)$ and *push* operations $\mathcal{O}(m \cdot n^2)$. Execution of each instance of an operation takes $\mathcal{O}(1)$ time; so the algorithm runs in $\mathcal{O}(m \cdot n^2)$ time.

Let cah be the cumulative active nodes' heights, $(+u : u \text{ active} : h(u))$. During this analysis, I derive how each operation affects the value of cah , which is needed in establishing the time bounds.

6.12.3.1 Effect of *relabel*

As I have shown in proving termination, each execution of *relabel* increases the cumulative heights, ch , by at least 1. Since $h(u) < 2 \cdot n$ for any intermediate node u , $ch < 2 \cdot n \cdot (n - 2)$, so the number of *relabel* operations is $\mathcal{O}(n^2)$.

Further, *relabel* always increases cah because only active nodes get relabeled. Since $cah \leq ch$, the increase in cah is at most $2 \cdot n^2$.

6.12.3.2 Effect of Saturating *push*

Execution of *push*(u, v) is *saturating* if $c_f(u, v) = 0$ after its execution, *non-saturating* otherwise. I consider saturating and non-saturating pushes separately.

Consider two saturating push operations on edge $u \rightarrow v$ in an execution where $push_1(u, v)$ precedes $push_2(u, v)$. I show that $h(v)$ increases. Exercise 28 (page 371) shows that both $h(u)$ and $h(v)$ increase between these two operations.

Every *push*(u, v) has the precondition $c_f(u, v) > 0$. A saturating *push*(u, v) has the postcondition $c_f(u, v) = 0$. Therefore, I claim the following scenario between the two *push*(u, v):

$push_1(u, v) \{ c_f(u, v) = 0 \} \cdots \{ c_f(u, v) > 0 \} push_2(u, v)$.

Here I write $\cdots$ for the intervening steps of execution. Now, once $c_f(u, v) = 0$, it remains so until there is an execution of a *push*(v, u) that establishes $c_f(u, v) > 0$. Therefore, the execution is of the form:

$push_1(u, v) \{ c_f(u, v) = 0 \} \cdots push(v, u) \{ c_f(u, v) > 0 \} \cdots push_2(u, v)$.

A precondition and postcondition of each $push(u, v)$ is $h(u) > h(v)$. Applying this observation to $push_1(u, v)$ and the subsequent $push(v, u)$, we get:

$$push_1(u, v) \{ h(u) > h(v) \} \dots \{ h(v) > h(u) \} push(v, u) \dots push_2(u, v).$$

Therefore, v is relabeled during the execution between $push_1(u, v)$ and $push(v, u)$, increasing its height.

The number of times $h(v)$ can increase, from (I3), is at most $2 \cdot n$. So the number of saturating $push(u, v)$ operations in an execution is at most $2 \cdot n$. Counting all m edges for $u \rightarrow v$, the number of saturating pushes is at most $2 \cdot m \cdot n$.

Note The argument shown above to establish that $h(v)$ increases in the intervening execution between two $push(u, v)$ cannot be formalized in the proof theory described in Chapter 4. Such an argument requires temporal logic for its formalization. ■

Effect of saturating push on cah I calculate how cah is affected by all saturating push operations; this analysis is needed for computing the number of non-saturating pushes.

The relevant precondition and postcondition of $push(u, v)$ are shown below.

$$\{ u \text{ active} \} push(u, v) \{ v \text{ active} \}$$

Thus, cah can increase by at most the value of $h(v)$ as a result of $push(u, v)$ (the maximum increase in cah happens if u remains active and v becomes active.). From (I3), $h(v) < 2 \cdot n$, and, since the number of saturating pushes is at most $2 \cdot m \cdot n$, all saturating pushes can increase cah by at most $4 \cdot m \cdot n^2$.

6.12.3.3 Effect of Non-saturating push

In any $push(u, v)$, the amount sent from u to v is $\min(f_{in}(u), c_f(u, v))$, and for a non-saturating push this quantity is less than $c_f(u, v)$. So, the entire inflow of u is sent to v , and $f_{in}(u) = 0$ after the operation, that is, u is no longer active. The effect is shown in

$$\{ u \text{ active}, h(u) > h(v) \} push(u, v) \{ u \text{ inactive}, v \text{ active}, h(u) > h(v) \}$$

Since u becomes inactive and $h(u) > h(v)$, cah decreases by at least 1 (it decreases by exactly 1 if v was inactive prior to the operation).

I use the effects of various operations on cah to count the number of non-saturating pushes. The initial value of cah is 0 because all active nodes have 0 height and the final value of cah is 0 because there is no active node.

$$\begin{aligned} & \text{the number of non-saturating push} \\ \leq & \{ \text{each non-saturating push decreases } cah \text{ by at least } 1 \} \\ & \text{the decrease in } cah \text{ due to non-saturating pushes} \\ = & \{ \text{initial and final values of } cah \text{ are equal} \} \end{aligned}$$

$$\begin{aligned}
& \text{the increase in } cah \text{ due to relabel and saturating pushes} \\
\leq & \{ \text{increase in } cah \text{ due to relabel is } \leq 2 \cdot n^2 \\
& \text{increase in } cah \text{ due to saturating pushes } \leq 4 \cdot m \cdot n^2 \} \\
& 2 \cdot n^2 + 4 \cdot m \cdot n^2
\end{aligned}$$

Complexity of Goldberg-Tarjan algorithm Combining the number of *relabel*, $\mathcal{O}(n^2)$, saturating push, $\mathcal{O}(m \cdot n)$, and non-saturating push, $\mathcal{O}(m \cdot n^2)$, the complexity of Goldberg-Tarjan algorithm is $\mathcal{O}(m \cdot n^2)$. Variations of this algorithm, using specific heuristics in choosing $u \rightarrow v$ for $push(u, v)$ and more sophisticated data structures, achieve $\mathcal{O}(n^3)$ execution time.

6.13 Exercises

1. Show that in an undirected graph the number of nodes of odd degree is even.

Solution Let $oddN$ be the set of nodes of odd degree and $evenN$ of even degree. We have to show that $|oddN|$ is even. Let $deg(u)$ be the degree of node u . The sum of degrees of all nodes is:

$$\begin{aligned}
& (+u : u \text{ any node} : deg(u)) \\
= & \{ \text{divide nodes into odd and even nodes by degree} \} \\
& (+u : u \in oddN : deg(u)) + (+u : u \in evenN : deg(u))
\end{aligned}$$

The sum of the degrees of all the nodes counts each edge twice because an edge is incident on two nodes. So, $(+u : u \text{ any node} : deg(u))$ is even. The second term, $(+u : u \in evenN : deg(u))$, is even because it is a sum of even numbers. So, the first term, $(+u : u \in oddN : deg(u))$, is even. This term is a sum of $|oddN|$ odd numbers; so $|oddN|$ is even.

2. Show that in a directed or undirected graph if there is path between a pair of nodes there is a simple path, and similarly for cycles.

Hint Use induction on the length of path/cycle.

3. Show that a directed graph is strongly connected iff for any node x there is a path from x to every other node and a path from every other node to x .
4. In a directed graph a node is a *universal sink* if every node has an outgoing edge to it and it has no outgoing edge. Clearly, a graph can have at most one universal sink. Give an efficient algorithm to locate one, if it exists, given the adjacency matrix of the graph.

Solution Let c be a set of nodes that satisfies the invariant: every node outside c is *not* a universal sink. Initially, c is the set of all nodes. In each step, consider a pair of nodes u and v in c . If there is no edge from u to v , then v is

a not a sink because it has no incoming edge from u . And, if there is an edge from u to v , then u is not a sink because it has an outgoing edge. So, one of u and v can be removed from c . Continue removing nodes until c has just one node and then test that node for the universal sink property.

5. I have 80 postage stamps in a 10×8 sheet. I would like to separate them into individual postage stamps by cutting along the perforations. A cut along a horizontal or vertical perforation separates any rectangular sheet into two rectangular sheets. How many cuts do I need at minimum to separate all 80 stamps?

Solution Represent a rectangular sheet as a node in a tree. A cut produces two children. So, required cuts produce a binary tree that has 80 leaf nodes, each a 1×1 sheet, and non-leaf nodes with exactly two children each. Each non-leaf node accounts for one cut. The number of non-leaf nodes in such a tree, from property (4) of Section 6.3.1.1 (page 274), is $l - 1$, where l is the number of leaf nodes. So, 79 cuts are needed to separate a 10×8 sheet.

Observe that there is no optimal strategy; any sequence of 79 cuts suffices.

6. Show that if all the incoming edges to a sink in an acyclic graph have their directions changed, so they all become outgoing, the graph remains acyclic.

Hint Prove by contradiction that the resulting graph has no cycle.

Note This observation is the basis of an algorithm in Chandy and Misra [1984] for the dining philosophers problem in concurrent programming.

7. Construct an Euler cycle for the graph in Figure 6.18(c). Show that the following substrings appear in that cycle: 010, 111 and 001.
8. Consider the iterated logarithm function introduced in Section 6.7.2.4

(page 310). Show: $\log^* \in \mathcal{O}(\overbrace{\log \log \cdots \log}^k)$, for any fixed value of k .

Hint I use ${}^k\log$ for $\overbrace{\log \log \cdots \log}^k$ below; take ${}^0\log$ to be the identity function. First, prove by induction on k that ${}^k\log(t_i) = t_{i-k}$, for $i \geq k$, where t is the dual function defined in that section. Next, show that $\log^* n < {}^k\log(n)$ for all n , $n > t_{2,k+3}$.

9. Consider the transitive closure A^+ and the reflexive and transitive closure A^* of the adjacency matrix A defined in Section 6.8.1 (page 318). Let I be the identity matrix of the same size as A . Given $x + x = x$ (i.e., $+$ is idempotent) for any matrix element x , show (1) $A^* = (I + A)^{n-1}$, (2) $A^+ = A \times A^*$, (3) for any m , $m \geq n$, $A^m + (+i : 0 \leq i < n : A^i) = (+i : 0 \leq i < n : A^i)$, and (4)

$A^+ = (+j : 0 \leq 2^j < n : A^{2^j})$. Based on (4), show a computation procedure for A^+ and A^* that takes around $\Theta(n^3 \cdot \log n)$ time.

Computation of A^+ and A^* Compute the terms $A, A^2, A^4, \dots, A^{2^j}$, where $2^j < n$, each term by multiplying the previous term with itself. Compute A^+ using (4), and A^* as $I + A^+$. This strategy requires doing around $\Theta(\log n)$ matrix multiplications, and a traditional matrix multiplication takes $\Theta(n^3)$ time.

10. In connection with space requirement for Warshall's algorithm of Section 6.8.2.1 (page 320), I introduced proposition

$$P_t :: (\forall x, y :: x \overset{\leq t}{\rightsquigarrow} y \Rightarrow W[x, y] \Rightarrow x \rightsquigarrow y).$$

Prove that P_t is an invariant in two steps, outlined below.

- (a) Show the correctness of the following annotation.

$$\begin{aligned} & \{P_t\} \\ & W[u, v] := W[u, v] \vee (W[u, t] \wedge W[t, v]) \\ & \{P_t, u \overset{\leq t+1}{\rightsquigarrow} v \Rightarrow W[u, v] \Rightarrow u \rightsquigarrow v\} \end{aligned}$$

Hint For the proof, it is sufficient to prove only a part of the desired postcondition:

$$\begin{aligned} & \{(\forall x, y :: x \overset{\leq t}{\rightsquigarrow} y \Rightarrow W[x, y] \Rightarrow x \rightsquigarrow y)\} \\ & W[u, v] := W[u, v] \vee (W[u, t] \wedge W[t, v]) \\ & \{u \overset{\leq t+1}{\rightsquigarrow} v \Rightarrow W[u, v] \Rightarrow u \rightsquigarrow v\} \end{aligned}$$

We can show, from the annotation above, that P_t holds as a postcondition. This is because $W[x, y]$ is unchanged by the assignment for all $(x, y) \neq (u, v)$. And, from $u \overset{\leq t+1}{\rightsquigarrow} v \Rightarrow W[u, v] \Rightarrow u \rightsquigarrow v$ as a postcondition, $u \overset{\leq t}{\rightsquigarrow} v \Rightarrow W[u, v] \Rightarrow u \rightsquigarrow v$ holds as well because $u \overset{\leq t}{\rightsquigarrow} v \Rightarrow u \overset{\leq t+1}{\rightsquigarrow} v$ from the meaning of $u \overset{\leq t}{\rightsquigarrow} v$ (see Rule of consequence, Section 4.5.1, page 153).

- (b) Given this annotation, show that P_t is invariant in the main loop of Warshall's algorithm.

Hint From the annotation, P_t remains true as a precondition of each assignment during an iteration, and the assignment establishes the condition in P_{t+1} for (u, v) . So, after completion of the iteration, the condition in P_{t+1} holds for all (u, v) ; that is, P_{t+1} holds.

11. Consider the all-pair shortest path algorithm of Section 6.8.3 (page 321). Propose a modification to reduce its space requirement along the lines suggested in Exercise 10 for Warshall's algorithm.

Hint Replace $W_{t+1}[u, v] := \min(W_t[u, v], W_t[u, t] + W_t[t, v])$ in the algorithm by $W[u, v] := \min(W[u, v], W[u, t] + W[t, v])$. Use the invariant:

$$(\forall x, y :: x \rightsquigarrow^t y \geq W[x, y] \geq x \rightsquigarrow y).$$

12. Where does the all-pair shortest path algorithm of Section 6.8.3 (page 321) use the fact that the edge lengths are non-negative?

Solution The derivation for $W_{t+1}[u, v]$ assumes that any shortest path is a simple path. Then the shortest path from u to v that includes node t is of the form $u \rightsquigarrow^t t \rightsquigarrow^t v$, that is, there is a single occurrence of t on the path. If edge lengths are allowed to be negative, there may be a cycle including t whose total length is negative. Then any shortest path that includes t will include infinite repetition of the cycle because each time around the cycle lowers the path length.

This argument also shows that edge lengths may be negative as long as there is no cycle of negative length.

13. In a directed graph where each edge has non-negative length, let the *distance* from node u to v be the length of the shortest path from u to v . Define the *span* of a node to be the maximum distance from this node to any node. A node is a *center* if it has the smallest span. Suggest an algorithm for locating a center.
14. (a) Show that $B = (\{true, false\}, \vee, \wedge, false, true)$ is a closed semiring. This semiring is used in Warshall's algorithm in Section 6.8.2 (page 319).
- (b) Show that $Reals^+ = (\text{non-negative reals} \cup \{\infty\}, \inf, +, \infty, 0)$ is a closed semiring, where *inf* is *infimum* over a set of reals. Since *min* is not defined over an infinite set of reals, we use *inf*, which is the greatest lower bound over a subset of reals, if it exists. This semiring is used in the all-pairs shortest path problem (Section 6.8.3, page 321).
- (c) Let $MetroSingle = (L, \cup, \cap, \{\}, L)$, where L is a finite alphabet. Show that $MetroSingle$ is a closed semiring. This semiring may be used for computing all routes without transfer points between pairs of stations in a metro.
15. (Converse of SP-Eqn) Consider a rooted directed graph that has no negative cycle. Associate $d(u)$, a non-negative value, with each node u so that (SP-Eqn) (Section 6.9.2, page 326) is satisfied.
- (a) Is $d(u)$ the distance from the root node to u for each u ?

Solution Consider a graph with nodes $root$, u and v with the following edge lengths, shown as labels on edges: $root \xrightarrow{10} u$, $u \xrightarrow{0} v$ and $v \xrightarrow{0} u$.

Let $d(root)$, $d(u)$, $d(v) = 0, 3, 3$. There is no negative cycle in the graph and the assigned values satisfy the (SP-Eqn). Yet these values are not the distances.

- (b) Show that $d(u)$ is the distance from the root node to u , for each u , if all edge lengths are positive.

Hint Use induction on ranks.

16. Consider the shortest path algorithm in Figure 6.48 (page 332). Show that for any unreachable node x from $root$, $td[x] = \infty$ at termination of the algorithm.
17. Given a shortest path algorithm for edge lengths that are positive real numbers, modify the algorithm to accommodate zero-length edges as well.

Solution Given that $x \rightarrow y$ has zero length, merge x and y as described in “Merging nodes”, Section 6.2.6.2 (page 272). Continue until all zero-length edges are eliminated. The order of merging is irrelevant. Determine the distances in the revised graph using the given algorithm. Suppose x , y and z were merged to form xyz . Then the distances of x , y and z are the same as that of xyz .

18. Given that all edge lengths are positive, consider the graph that contains only the edges of the shortest paths. What is its structure?

Hint Use induction on ranks to show that the structure is a rooted tree.

19. It is required that if the two shortest paths to a node have equal distance, then the one with fewer edges is to be chosen. (If there is still a tie, choose any one of these paths). Modify Dijkstra’s algorithm (Figure 6.48, page 332) to implement this requirement.

Hint Define path length as a tuple (c, d) where c is the sum of edge lengths along the path and d the number of edges. Compare path lengths lexicographically.

20. Consider the simulation-based shortest path algorithm of Section 6.9.5 (page 332).
 - (a) Is it possible for *event_queue* to have an entry (t, y) where y is *lit*?
 - (b) Show that as long as there is an unlit node *event_queue* is not empty. You will have to use the fact that every node is reachable from *root*.
 - (c) Derive an upper bound on the number of steps in the algorithm assuming that the simplest procedure is used for removing entries from *event_queue*.

21. Rewrite the inner loop of the Bellman–Ford algorithm (Figure 6.50, page 338) using edge relaxation as follows, and prove its correctness. Below E is the set of edges.

for $u \rightarrow v \in E$ **do** $td[v] := \min(td[v], td[u] + e(u, v))$ **endfor**

Observe that the invariant, $I :: (\forall v :: td[v] = d_i(v))$, no longer applies because $u \rightarrow v$ may be relaxed before or after $td[u]$ has been updated during an iteration. As in the implementation of Warshall’s algorithm (Section 6.8.2), use invariant I' for the proof:

$$I' :: (\forall v :: d_i(v) \leq td[v] \leq d(v)).$$

22. (Minimum spanning tree) Show that if all edges of the graph have distinct lengths, then the mst is unique.

Hint Show that Kruskal’s algorithm (Section 6.10.3, page 345) constructs a unique mst for such a graph.

23. (Minimum spanning tree) This exercise generalizes Theorem 6.13 (page 342). Let M be a subset of some mst and E a set edges such that each edge is safe for exactly one component. Then $M \cup E$ is a subset of some mst.

Observe that edges in E may have equal lengths as long as they are safe for different components.

Hint Prove Lemma 6.5 (page 342) under the given condition. Then verify that the proof of Theorem 6.13 is still valid.

24. (Minimum spanning tree) Consider the following algorithm for computing a minimum spanning tree in a graph with distinct edge lengths. Initially, the mst M is empty. In each step add a shortest edge $x-y$ to M , and then modify the graph by merging nodes x and y , as described in Section 6.2.6 (page 271). Repeat the steps until the graph has no edges. Prove that at termination M is a minimum spanning tree. Because of distinct edge lengths, the edge that is being added to M can be identified in the original graph.

Hint Show that this is a rewriting of Kruskal’s algorithm. Call the graph at any point a *reduced graph*. Prove the invariant that each component formed by M is a node in the reduced graph. Therefore, each edge in the reduced graph is an inter-component edge. Cycle detection is replaced by the merging step so that no edge of the reduced graph forms a cycle with M at any point during the execution.

25. (Minimum spanning tree)

- (a) Show that in a graph in which all edge-lengths are distinct, the mst (which is unique) does not contain the maximum-length edge of any cycle.

Solution Let c be the edges of a cycle of which e is the maximum-length edge. Let M be an mst to which e belongs. Not all edges of c belong M because c is a cycle; suppose $f \notin M$ for an edge f of c . Then $M - \{e\} \cup \{f\}$ is a spanning tree whose length is smaller than that of M (see Lemma 6.4 (page 340), properties (ST3, ST4)).

- (b) Vic Vyssotsky, formerly of Bell Labs, proposed the following algorithm for computing an mst, in an unpublished note around 1962. Start with an empty set M for the mst. Pick any edge e that has not been picked earlier and add it to M . If e forms a cycle with M , remove a maximum-length edge, f , of the cycle from M , so $M := M \cup \{e\} - \{f\}$. Note that e and f could be identical. Continue until all edges have been picked. At termination, M is an mst.

This can be regarded as a streaming algorithm where the edges are generated in arbitrary order in a stream, and the computation of the mst is completed by the time the stream ends.

Prove the correctness of this algorithm.

Hint Prove that M is an mst over the edges picked at any point. Use the property in part (a) of this exercise.

26. (Maximum flow) This exercise asks for a proof of part (3) of Lemma 6.6 (page 354). For cut (S, T) of a capacitated network, $f_{out}(S) = f_{out}(s) = f_{in}(t) = f_{in}(T)$.

In particular, $|f| = f(S, T)$.

Solution

Proof of $f_{out}(S) = f_{out}(s)$:

$$\begin{aligned}
 & f_{out}(S) \\
 = & \{S = \{s\} \cup (S - \{s\}). \text{ Apply distributivity} \} \\
 & f_{out}(s) + f_{out}(S - \{s\}) \\
 = & \{\text{from (F1), } f_{out}(x) = 0 \text{ for } x \text{ in } (S - \{s\}). \text{ So, } f_{out}(S - \{s\}) = 0\} \\
 & f_{out}(s)
 \end{aligned}$$

Proof of $f_{in}(T) = f_{in}(t)$: similarly.

Proof of $f_{out}(S) = f_{in}(T)$:

$$\begin{aligned}
 & V = S \cup T \\
 \Rightarrow & \{\text{apply distributivity}\} \\
 & f_{out}(V) = f_{out}(S) + f_{out}(T) \\
 \Rightarrow & \{f_{out}(V) = f(V, V) = 0\}
 \end{aligned}$$

$$\begin{aligned}
0 &= f_{out}(S) + f_{out}(T) \\
\Rightarrow \{f_{out}(T) &= -f_{in}(T)\} \\
f_{out}(S) &= f_{in}(T)
\end{aligned}$$

To show that $|f| = f(S, T)$: $|f| = f_{out}(s)$, and we just proved $f_{out}(s) = f_{out}(S)$. Show $f_{out}(S) = f(S, T)$.

27. (Goldberg–Tarjan algorithm)

Prove that after initialization there is never a positive path from s to t .

28. (Goldberg–Tarjan algorithm)

Consider two saturating push operations, $push_1(u, v)$ and $push_2(u, v)$, on edge $u \rightarrow v$ where $push_1(u, v)$ precedes $push_2(u, v)$ in an execution. Show that both $h(u)$ and $h(v)$ increase between these two operations.

Solution

From Section 6.12.3.2 (page 362), we have the following execution scenario between two saturating pushes on an edge.

$push_1(u, v) \{h(u) > h(v)\} \dots \{h(v) > h(u)\} push(v, u) \dots push_2(u, v)$.

I showed that $h(v)$ increases between $push_1(u, v)$ and $push(v, u)$. Switching the roles of u and v , $h(u)$ increases between $push(v, u)$ and $push_2(u, v)$.

6.A Appendix: Proof of the Reachability Program

I prove the correctness of the abstract program for reachability in Figure 6.20 (page 291). The program is annotated in Figure 6.65 using the invariant, Inv , proposed there:

$$Inv:: root \cup succ(marked) \subseteq marked \cup vroot \subseteq R(\{root\}).$$

The assertion Inv holds after initialization, by inspection. The proof of the post-condition at termination, $(Inv \wedge vroot = \{\}) \Rightarrow (marked = R(\{root\}))$, has been given in Section 6.5.2 (page 291). The remaining verification condition is: $Inv' \Rightarrow Inv''$.

$$\begin{aligned}
&root \cup succ(marked) \subseteq marked \cup vroot \subseteq R(\{root\}), u \in vroot \\
\Rightarrow & \\
&root \cup succ(marked \cup \{u\}) \\
&\subseteq marked \cup \{u\} \cup ((vroot \cup succ(\{u\}) - (marked \cup \{u\})) \\
&\subseteq R(\{root\})
\end{aligned}$$

First, simplify

$$\begin{aligned}
&marked \cup \{u\} \cup ((vroot \cup succ(\{u\}) - (marked \cup \{u\})) \\
&marked \cup \{u\} \cup vroot \cup succ(\{u\}).
\end{aligned}$$

Abstract program for reachability

```

marked, vroot := {}, {root};
{ Inv }
while vroot ≠ {} do
{ Inv, vroot ≠ {} }
  u ∈ vroot ;
{ Inv' :: root ∪ succ(marked) ⊆ marked ∪ vroot ⊆ R({root}), u ∈ vroot }

{ Inv'' :: root ∪ succ(marked ∪ {u})
  ⊆ marked ∪ {u} ∪ ((vroot ∪ succ({u}) − (marked ∪ {u}))
  ⊆ R({root}) }

marked := marked ∪ {u};
vroot := (vroot ∪ succ(u)) − marked

{ Inv :: root ∪ succ(marked) ⊆ marked ∪ vroot ⊆ R({root}) }
enddo
{ Inv ∧ vroot = {} }
{ marked = R({root}) }

```

Figure 6.65

I prove the two parts of *Inv''* below.

- *root* ∪ *succ*(*marked* ∪ {*u*}) ⊆ *marked* ∪ {*u*} ∪ *vroot* ∪ *succ*({*u*}):

$$\begin{aligned}
& \text{root} \cup \text{succ}(\text{marked} \cup \{u\}) \\
&= \{\text{succ}(\text{marked} \cup \{u\}) = \text{succ}(\text{marked}) \cup \text{succ}(\{u\})\} \\
& \quad \text{root} \cup \text{succ}(\text{marked}) \cup \text{succ}(\{u\}) \\
&\subseteq \{\text{from } \text{Inv}, \text{root} \cup \text{succ}(\text{marked}) \subseteq \text{marked} \cup \text{vroot}\} \\
& \quad \text{marked} \cup \text{vroot} \cup \text{succ}(\{u\}) \\
&\subseteq \{\text{trivially}\} \\
& \quad \text{marked} \cup \{u\} \cup \text{vroot} \cup \text{succ}(\{u\})
\end{aligned}$$

- *marked* ∪ {*u*} ∪ *vroot* ∪ *succ*({*u*}) ⊆ *R*({*root*}):

$$\begin{aligned}
& \text{marked} \cup \{u\} \cup \text{vroot} \cup \text{succ}(\{u\}) \\
&\subseteq \{\text{from } \text{Inv}, \text{marked} \cup \text{vroot} \subseteq \text{R}(\{\text{root}\})\} \\
& \quad \{u\} \cup \text{succ}(\{u\}) \cup \text{R}(\{\text{root}\}) \\
&\subseteq \{\text{from graph theory}, \{u\} \cup \text{succ}(\{u\}) \subseteq \text{R}(\{u\})\}
\end{aligned}$$

$$\begin{aligned}
& R(\{u\}) \cup R(\{\text{root}\}) \\
= & \{u \in \text{root}. \text{From } \text{Inv}, \text{root} \subseteq R(\{\text{root}\}), \text{ so } u \in R(\{\text{root}\})\}. \\
& R(\{u\}) \subseteq R(R(\{\text{root}\})) = R(\{\text{root}\}), \text{ from (P4) in Section 6.5.1} \\
& R(\{\text{root}\})
\end{aligned}$$

6.B

6.B.1

Appendix: Depth-first Traversal

Appendix: Proof of Edge-ancestor Lemma

Lemma 6.2 (page 301) For edge $u \rightarrow v$: $u_e < v_e \equiv v$ is ancestor of u .

Proof. In one direction, v is ancestor of $u \Rightarrow u_e < v_e$, the proof follows from the properties of postorder numbering, Proposition 6.1 (page 298). I prove the lemma in the other direction, for edge $u \rightarrow v$: $u_e < v_e \Rightarrow v$ is ancestor of u .

In Figure 6.66 (page 374), the depth-first traversal program is reproduced from Figure 6.27 (page 300) with added annotations. For marked nodes x and y , $x \text{ anc } y$ means that x is an ancestor of y , as given by the *parent* relation. Any successor v of u for which $\text{dfs}(v)$ has completed is an “explored-successor” of u . The nodes of the tree rooted at u , as given by the *parent* relation, are assigned preorder and postorder numbers in the usual fashion. Also, $x.\text{mark}$ and $x.\text{done}$ remain true once they have been assigned, and the *parent* of a node never changes, so once $x \text{ anc } y$ holds it continues to hold.

$$\begin{aligned}
I &:: (\forall x :: x.\text{done} \Rightarrow x.\text{mark} \wedge x_e < e_e) \\
p(u) &:: (\forall x :: x.\text{mark} \Rightarrow x.\text{done} \vee x \text{ anc } u) \\
q(u) &:: u.\text{done} \wedge (\forall y \rightarrow z, u \text{ anc } y :: z.\text{done} \wedge z_e < y_e \vee z \text{ anc } y)
\end{aligned}$$

I omit a number of easy proofs, in particular that $u.\text{done}$, $v_e < u_e$, $v \text{ anc } u$ remain true once they are true. Also, invariant I is easy to establish; I do not show its proof. I show

$$\{p(u)\} \text{dfs}(u) \{p(u), q(u)\}.$$

Below V is the set of all nodes.

The postcondition of $\text{dfs}(\text{root})$ is justified as follows. Predicate $q(u)$ remains true once it is true because each of the individual terms in it has this property. For every node u reachable from $\text{dfs}(\text{root})$, $\text{dfs}(u)$ is executed. So, the postcondition of $\text{dfs}(u)$, for every u , holds after termination of $\text{dfs}(\text{root})$. Hence, for every $u \rightarrow v$ and $u_e < v_e$, conclude

$$(v.\text{done} \wedge v_e < u_e \vee v \text{ anc } u) \wedge u_e < v_e, \text{ which implies } v \text{ anc } u.$$

Depth-first traversal of directed graph, Annotated

```

Proc dfs(u)
  {p(u)}
  ub, ecb, u.mark := ecb, ecb + 1, true;
  {p(u), given u.mark and u anc u}
  for v ∈ succ(u) do
    {p(u)}
    if v.mark then skip {p(u)}
    else
      parent[v] := u ;
      {p(v), from p(u) and ancestors of u are ancestors of v}
      dfs(v)
      {p(v), q(v)}    {p(v)}
      {(∀x :: x.mark ⇒ x.done ∨ x = v ∨ x anc u)}
      {p(u), given v.done, x = v ⇒ x.done}
    endif
  {p(u)}
  endfor
  {p(u), (∀x :: x.mark ⇒ xe < ece), using I}
  ue, ece, u.done := ece, ece + 1, true
  {p(u), u.done, (∀x :: x.mark ⇒ (x.done ∧ xe < ue) ∨ x anc u)}
  {p(u), u.done, all v in succ(u) are marked, So (∀u → v :: (v.done ∧ ve < ue) ∨ v anc u)}
  {p(u), q(u)}
endProc

for u ∈ V do u.mark, u.done := false, false endfor;
  {(∀u :: ¬u.mark, ¬u.done)}
  ecb, ece := 0, 0;
  {p(u)}
  dfs(root)
  {(∀u :: q(u))}
  {(∀u → v :: ue < ve ⇒ v anc u)}

```

Figure 6.66

6.B.2 Appendix: Proof of Path-ancestor Theorem

Theorem 6.8 (page 302) Given $u \xrightarrow{p} v$, where $u_e < v_e$, the highest node in p is the lowest common ancestor of all nodes in p .

Proof. It suffices to show that the highest node is an ancestor of all nodes in p , and the theorem follows because the highest node is an ancestor of itself. The theorem is valid even for a path that is not simple.

Let h be the highest node in p . Then the path is of the form $u \xrightarrow{lp} h \xrightarrow{rp} v$, where the first occurrence of h is as an extreme node in lp . I prove the result in two parts, that h is an ancestor of nodes in lp (Lemma 6.7) and in rp (Lemma 6.8).

Lemma 6.7 Given $u \xrightarrow{lp} h$ where h is the unique highest node in lp , h is an ancestor of all nodes in lp .

Proof. Proof is by induction on the length of lp .

(1) Base case, lp has one edge: then $u \rightarrow h$ and $u_e < h_e$. From edge-ancestor lemma, Lemma 6.2 (page 301), h is an ancestor of u . Since h is an ancestor of itself, the result holds.

(2) Inductive case, $u \rightarrow u' \xrightarrow{lp'} h$: Inductively, h is an ancestor of all nodes in lp' including u' . If $u_e < u'_e$, from edge-ancestor lemma, Lemma 6.2, u'_e is an ancestor of u , hence h is an ancestor of u . If $u'_e < u_e$, then $u'_e < u_e < h_e$ and h is an ancestor of u' . From the Convexity rule, Lemma 6.1 (page 298), h is an ancestor of u . ■

Lemma 6.8 Given $u \xrightarrow{lp} h \xrightarrow{rp} v$ where h is the highest node, h is an ancestor of all nodes in rp including v .

Proof. Note that all nodes of lp and rp , including h , may occur multiple times in rp . Assign consecutive positive indexes to the nodes in rp , from v to h , starting with index 1 for v . I show that h is an ancestor of the node of index k , for all k where $k \geq 1$. Proof is by induction on k . From Lemma 6.7, (page 375) h is an ancestor of u .

(1) Base case, $k = 1$: We have $u_e < v_e \leq h_e$ and h is an ancestor of u . Applying the Convexity rule, Lemma 6.1 (page 298), h is an ancestor of v .

(2) Inductive case, $k > 1$: Let w be the node of index k , so the path is $u \xrightarrow{lp} h \xrightarrow{rp'} w \xrightarrow{rp''} v$. If $w = u$, then h is an ancestor of w because h is an ancestor of u . So, assume $w \neq u$. Consider two cases.

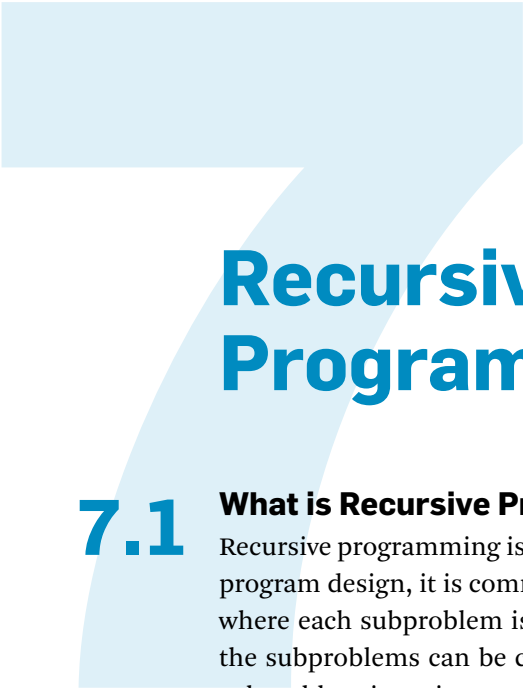
(2.1) $u_e < w_e$: Then $u_e < w_e \leq h_e$. Applying the Convexity rule, Lemma 6.1 (page 298), h is an ancestor of w .

(2.2) $u_e > w_e$: Then $w_e < u_e < v_e$. Let h' be the highest node in rp'' . Its index is less than k because, from $w_e < v_e \leq h'_e$, $h' \neq w$. Inductively, h is an ancestor of h' . Apply Lemma 6.7 (page 375) on $w \rightsquigarrow h'$ to conclude that h' is an ancestor of w . Therefore, h is an ancestor of w . ■

6.B.3 Appendix: Additional Properties of Depth-first Traversal

To establish the type of each edge during depth-first traversal, use the following invariant in the proof of $dfs(u)$. Below, $x \text{ ind } y$ means x and y are independent, that is, $(\neg x \text{ anc } y) \wedge (\neg y \text{ anc } x)$.

$$\begin{aligned}
 &u \neq \text{root} \Rightarrow \text{parent}[u] \rightarrow u, (\text{root} \text{ anc } u), \\
 &(\forall x \rightarrow y : u \text{ anc } x, y.\text{mark} : \\
 &\quad x_b < y_b \Rightarrow (x \text{ anc } y) \wedge y.\text{done} \wedge (\neg x.\text{done} \vee x_e > y_e) \wedge \\
 &\quad \quad \{ \text{forward or tree edge} \} \\
 &\quad x_b > y_b, y.\text{done}, (\neg x.\text{done} \vee x_e > y_e) \Rightarrow (x \text{ ind } y) \wedge \{ \text{cross edge} \} \\
 &\quad x_b > y_b, \neg y.\text{done} \Rightarrow (y \text{ anc } u) \{ \text{back edge} \} \\
 &)
 \end{aligned}$$



Recursive and Dynamic Programming

7.1 What is Recursive Programming

Recursive programming is closely tied to *problem solution through decomposition*. In program design, it is common to divide a problem into a number of subproblems where each subproblem is easier to solve by some measure, and the solutions of the subproblems can be combined to yield a solution to the original problem. A subproblem is easier to solve if its solution is known or if it is an instance of the original problem but over a smaller data set. For example, if you have to sum eight numbers, you may divide the task into two subtasks of summing four numbers each, and then add the two results. Each subtask may be further divided into two subtasks of adding a pair of numbers.

A problem is typically divided into subproblems of the same kind in recursive programming, and the same solution procedure is applied to each of the subproblems, further subdividing them. A recursive program has to specify the method of problem decomposition and solution combination and also the solutions for the very smallest subproblems that cannot be decomposed any further.

The theoretical justification of recursive programming is *mathematical induction*. In fact, recursion and induction are so closely linked that they are often mentioned in the same breath; we ought to have used the term “inductive programming” for this style of programming.

Patterns in computation A programming technique is effective if it can encode patterns of computation that arise often in practice. A Turing machine is a simple and elegant tool for describing all computations. Yet we do not adopt it as a programming language because it provides little aid in describing common computational patterns or data structures. Over the years, we have adopted programming languages with high-level features that can encode common programming patterns succinctly, thereby relieving the programmer from tasks that can be carried out more effectively by a computer.

Recursive programming is an outstanding example of making the machines more effective, as Dijkstra had prescribed as a goal of research in programming, see Section 1.2 (page 2). Recursive programs can encode extraordinarily intricate computation patterns, as we will see in this chapter¹.

Often, a problem is first solved by a recursive program and then hand-translated to an imperative program for reasons of efficiency. This translation is very simple if the program is tail recursive (see Section 7.2.5.7, page 391). Even if the program is not tail recursive, it is still possible to extract an efficient imperative program from it. In particular, dynamic programming (Section 7.6, page 432) is a general technique for efficient implementation of a class of recursive programs.

Functional programming and recursion A style of programming known as *functional programming* is especially suited for describing recursion in computations and data structures. Pure functional programs are closer to mathematical equations in their structure, which make them better amenable to mathematical analysis. They are often significantly more compact, and easier to design and understand, than their imperative counterparts. Induction is an essential tool in designing and reasoning about functional programs.

I adopt a programming notation derived from Haskell [see Marlow 2010], an elegant functional programming language. The notation covers a very small part of Haskell, only what is needed for explaining recursive programming. An excellent introduction, though somewhat dated, is Hudak et al. [2000]. The Haskell Prelude contains many excellent examples of function definition; also see a very good tour of Haskell Prelude at Pope [2001]. All the program snippets in this chapter were run using the Haskell interactive interpreter, `ghci`.

It should be clearly understood that recursion transcends functional programming. Recursion was introduced in programming languages in the early 1960s, in the imperative language Algol-60 and functional language Lisp. Common imperative languages now include recursion. I have shown several examples of recursive programs in imperative notation in the previous chapters of this book. In particular, see quicksort in Section 5.4 (page 208) and depth-first traversal in Section 6.6.2 (page 296). Both algorithms make use of mutable store that is unsuited for pure functional programming.

1. Tony Hoare, the inventor of quicksort (see Section 5.4, page 208), could not program the algorithm in any imperative language available to him at that time even though he understood it in recursive terms. He could program it only after he learned the Algol-60 programming language that included recursive constructs.

Functional programs eschew many other notions of imperative programming: besides mutable store, pointers, mechanisms for iteration and random access to data are not available. While iteration can mostly be coded as *tail recursion*, absence of random access and pointers make certain programming tasks much harder or impossible to implement efficiently as a functional program. A notable example is quicksort, which is inherently recursive though it modifies array elements in place; it is not possible to encapsulate its essence in a pure functional program. Even if an imperative programming feature can be simulated in a purely functional style, one has to pay a heavy penalty in resource usage. Therefore, functional programs should be used only for what they are best suited for.

Note on font usage This chapter is unusual in that I show a large number of Haskell program snippets. Each snippet is in typewriter font, for example, `fibpair`, to make it easily distinguishable. For an interactive session with `ghci`, the human input is in blue color and the computer output in black, though both are in typewriter font. Unfortunately, this distinction is lost in a black and white printed copy.

I use mathematical font, for example, *fibpair*, for proofs. Also, I use `=` in its mathematical sense in proofs, avoiding `==` which is used for equality tests in Haskell.

Programs written in imperative style in Section 7.6 (page 432) follow the font and color usage conventions of the previous chapters: **boldface** for keywords and blue color for tests and assignments, and black and white for assertions.

7.2 Programming Notation

Prefix and infix operators An *operator* is a function. Application of an operator on its *operands* returns a value. In this chapter, I reserve *operator* to mean a *binary operator*, a function over two operands. An *infix operator*, such as `+`, is written between its two arguments, whereas a *prefix operator*, such as `max`, precedes its arguments. Convert an infix operator to a prefix operator by putting parentheses around the function name (or symbol). Thus, `(+) x y` is the same as `x + y`; convert from prefix to infix by putting backquotes around an operator, so `div 5 3` is the same as `5 `div` 3`.

Built-in binary operators in Haskell that begin with a letter, such as `max`, `min`, `rem`, `div` and `mod`, are prefix operators, that is, they precede their operands in an expression. Operators that do not begin with a letter, such as `+`, `*`, `&&` and `||`, are typically binary infix operators, that is, they are written between their two operands.

7.2.1 Basic Data Types

I use the following basic types of Haskell: `Int` for short integers,² `Float` for floating point numbers, `Bool` for booleans, `Char` for a single character and `String` for character strings.

Int and Float The usual arithmetic operations, addition (+), subtraction (-) and multiplication (*), written as infix operators, are supported for `Int` and `Float` types. Unary minus sign is the usual -, but it is always safe to enclose a negative number within parentheses, as in (-2), because of Haskell's conventions about function binding (see the binding rules in Section 7.2.3, page 383). The prefix operator `div` for integer division returns only the quotient, the integer part of the result of division. Thus, `div 9 3` is 3 and `div 9 2` is 4. The remainder after integer division is given by the prefix operator `rem`. The prefix operator `mod` is for modulo; `mod x y` returns a value between 0 and y-1 for a positive integer y. Exponentiation is the infix operator ^, so that 2¹⁵ is 32768. Division over floating point numbers is written using the infix operator /.

Two other useful functions are `even` and `odd`, which return the appropriate boolean results about their integer arguments. Prefix operators `max` and `min` take two arguments each and return the maximum and the minimum values, respectively.

The arithmetic relations are < <= == /= > >=. Each of these is a binary infix operator that returns a boolean result, `True` or `False`. Equality operator is written as == and inequality as /=.

Bool There are the two boolean constants, written as `True` and `False`. The boolean operators are: `not`, `&&`, `||`, `==` and `/=`, for negation, logical and, logical or, equivalence and inequivalence (or exclusive or), respectively. Here is a short interactive session with `ghci`.

```
Prelude> (3 > 5) || (5 > 3)
True
Prelude> (3 > 3) || (3 < 3)
False
Prelude> (2 `mod` (-3)) == ((-2) `mod` 3)
False
Prelude> even(3) || odd (3)
True
```

2. Haskell supports two kinds of integers, `Integer` and `Int` data types. I use only `Int`. I do not address the problem of overflow in arithmetic.

Character and String Haskell supports Unicode characters and strings over them. A character is enclosed within single quotes, as in 'a', and a string within double quotes, as in "a b c".

There is an internal table that stores all the characters supported by Haskell. Assume that the table entries are indexed 0 through 255, and each character is assigned a unique index in the table, though not all indices in this range correspond to characters on a keyboard. The order of the characters in this table is important for programming. The digits '0' through '9' are stored in the standard order and contiguously in the table. The letters of the Roman alphabet are also stored in the standard order and contiguously. All the digits precede all the uppercase letters, which precede all the lowercase letters.

There are two functions defined on characters, `ord` and `chr`. Function `ord(c)` returns the value of character `c` in the internal table; it is a number between 0 and 255. Function `chr` converts a number between 0 and 255 to the corresponding character. Therefore, for all characters `c`, `chr(ord(c)) == c`, and `ord(chr(i)) == i`, for all `i`, $0 \leq i < 256$. To use `ord` and `chr` in a program, first execute `import Data.Char`, which is a module in standard Haskell.

Letters and digits can be compared using arithmetic relations, so '3' < '5', 'A' < 'B' and 'a' < 'b'. More generally, '0' < ... < '9' < 'A' < ... < 'Z' < 'a' < ... < 'z'.

A string is a *list* of characters; all the rules about lists, which are described later in this chapter, apply to strings. In particular, strings are compared lexicographically using arithmetic comparison operators, so "A3" < "b9".

Variables A command of the form `x = 5` defines that variable `x` is equal to 5. Henceforth, `x` and 5 can be used interchangeably throughout the program. Unlike in imperative programming, a variable has one fixed value throughout its lifetime; so they are *immutable*. Equation `x = 5` should not be viewed as “assigning” 5 to `x`; instead, `x` is another name for the constant that is the value of the expression in its right side. It is illegal to write `x = x+1` because there is no value that satisfies this equation.

The right side of an equation is a functional expression over constants and other defined variables, as in `x = (2 `mod` (-3)) == ((-2) `mod` 3)` or `offset = chr(ord('a') - ord('A'))`.

Comments Any text following `--` in a line is considered a comment. This is a good way to introduce a short comment. For longer comments enclose a region within `{-` and `-}`, as shown below. Such comments can be nested.

```

{-
    Any string following -- in a line is a comment.
    This is a good way to introduce a short comment.
    For comments of several lines enclose the region
    within {- -}, as I am doing here.
-}

```

Nesting of comments is quite useful during program development because a block of code can be commented out by enclosing it within `{-` and `-}` no matter what is within the block.

Another useful trick to comment and uncomment blocks during development is shown below. Use `{--` and `--}` on separate lines to comment a block.

```

{--
block
--}

```

Now add a single `}` at the end of the first line, as shown below, to uncomment the block. The first line is a comment because `{--}` starts and ends a comment, and the last line starts with the string `--`, which makes it a comment.

```

{--}
block
--}

```

7.2.2 Data Structuring: Tuple, List

Haskell supports *tuple* and *list* for structuring data.

7.2.2.1 Tuple

Tuple is Haskell's version of *record*. In the simplest case, form a 2-tuple, or a *pair*, from two pieces of data, as in `(3,5)`, `(3,3)`, `("old","hat")`, `(True,1)` and `(False,0)`. In `(3,5)`, 3 is the first and 5 the second component. The components can be of different types. So, a tuple can include a tuple as a component as in `(3,(3,5))` or `((3,5),(3,5))`. However, `(3,(3,5))` is not the same as `(3,3,5)` or `((3,5),(3,5))`. A tuple can have any number of components greater than 1.

Functions `fst` and `snd` extract the first and second components of a pair, respectively. So, `fst(3,'a')` is 3 and `snd(3,'a')` is 'a'.

7.2.2.2 List

Each tuple has a fixed finite number of components. In order to process larger amounts of data, where the number of data items may not be known *a priori*,

Haskell uses the data structure *list*. A list consists of a finite³ sequence of items *all of the same type*; I explain types in greater detail in Section 7.2.6 (page 391). Here are some examples of lists.

```
[ ]                -- empty list
[1,3,5,7,9]        -- odd numbers below 10
[2,3,5,7]           -- all primes below 10
[[2],[3],[5],[7]]  -- a list of lists
[(3,5),(3,8),(3,5),(3,7)] -- a list of tuples
[[ (3,5), (3,8) ], [ (3,5), (3,7), (2,9) ]] -- a list of list of tuples
['a','b','c']      -- a list of characters
["Socrates", "Plato"] -- a list of strings
```

The following are not lists because not all their elements are of the same type.

```
[[2],3,5,7]
[(3,5), 8]
[(3,5), (3,8,2)]
['J',"misra"]
```

The order and number of elements in a list matter. So, $[2,3] \neq [3,2]$, $[2] \neq [2,2]$ and $[] \neq [[]]$. Unlike a set, elements may be repeated in a list, so a list encodes a bag.

The first element of a non-empty list is its *head* and the remaining list, possibly empty, its *tail*. For $[2,3,5,7]$, $[[2],[3],[5],[7]]$ and $["Socrates", "Plato"]$, the heads are, respectively, 2, $[2]$ and "Socrates" and the tails are $[3,5,7]$, $[[3],[5],[7]]$ and $["Plato"]$.

A list whose head is x and tail is xs is written as $x:xs$. The symbol $:$ is called *Cons*; cons is right associative, so $x:y:z = x:(y:z)$. Therefore, $[2,3,5]$, $2:[3,5]$, $2:3:[5]$ and $2:3:5:[]$ are all equal.

Naming convention for items and lists A common naming convention is to add an s at the end of an item name to denote a list of items of the same type. For example, if x is the head of a list, the tail is usually called xs , so the entire list is $x:xs$. A list of lists may be written as $(x:xs):xss$.

7.2.3 Function Definition

In defining a function in Haskell, the first point to note is how to write its arguments. Normally, we enclose the arguments of a function within parentheses. But

3. I deal only with finite lists in this book. Haskell permits definitions of infinite lists and computations on them, though a computation can produce only a finite portion of a list in finite time.

functions are so ubiquitous in Haskell, and functional programming in general, that the number of parentheses would be prohibitive. Instead, Haskell does not always require the use of parentheses but relies on text indentations and binding rules to disambiguate the syntax of expressions. Parentheses can still be used, but they can be largely avoided. As an example, `max x y` is the maximum of its two arguments, and similarly `min x y`.

7.2.3.1 Binding Rules

Consider the expression `f 1+1` where `f` is a function of one argument. In normal mathematics, this will be an invalid expression, and, if forced, you will probably interpret it as `f(1+1)`. In Haskell, this is a valid expression, and it stands for `(f 1)+1`, which is `f(1)+1`.

An expression consists of functions, operators and operands, as shown in the expression `-2 + sqr x * min 2 7 - 3`. Here, the first minus (called unary minus) is a prefix operator, `sqr` is a defined function of one argument, `+` and `*` are infix binary operators, `min` is a pre-defined Haskell function of two arguments, and the last minus is a binary infix operator.

Functions, such as `sqr` and `min`, bind more tightly than infix operators. Function arguments are written immediately following the function name, and the right number of arguments are used up for each function, for example, one for `sqr` and two for `min`. No parentheses are needed unless the arguments themselves are expressions. So, for function `g` of two arguments, `g x y z` is `(g x y) z`. And `g f x y` is `(g f x) y`. As a good programming practice, do not ever write `f 1+1`; make your intentions clear by using parentheses, writing `(f 1)+1` or `f(1+1)`.

How do we read `-2 + sqr x * min 2 7 - 3`? After the functions are bound to their arguments, we get `-2 + (sqr x) * (min 2 7) - 3`. That is, we are left with operators only, and the operators bind according to their binding powers. Since operator `*` has higher binding power than `+`, this expression would be interpreted as `-2 + ((sqr 9) * (min 2 7)) - 3`. The first `-` is recognized as a unary minus, a function of one argument, whereas the last `-` is understood to be a binary minus; so the expression is `(-2) + ((sqr 9) * (min 2 7)) - 3`.

All occurrences of infix `+` and `-` in the resulting expression have the same binding power. Operators of equal binding power usually, but not always, associate to the left. Therefore, the previous expression, `(-2) + ((sqr 9) * (min 2 7)) - 3`, is interpreted as the expression `((-2) + ((sqr 9) * (min 2 7))) - 3` and `5 - 3 - 2` as `(5 - 3) - 2`. Operators in Haskell are either (1) associative, so that the order does not matter, (2) left associative, as in `5 - 3 - 2` which is `(5 - 3) - 2`, or (3) right associative, as in `2^3^5` which is `2^(3^5)`. When in doubt, parenthesize.

Unary minus, as in `-2`, is particularly problematic. To apply function `suc` to `-2`, never write `suc -2` because it means `(suc) - (2)`; instead, write `suc (-2)`. And `-5 `div` 3` is `-(5 `div` 3)` which is `-1`, but a slight rewriting, `(-5) `div` 3`, gives `-2`.

Terminology In computing, it is customary to say that a function “takes an argument” and “computes (or returns) a value”. A function cannot “do” anything; it simply *has* arguments and it *has* a value for each sequence of arguments. Yet, the computing terminology is so prevalent that I will use these phrases without apology in this book.

7.2.3.2 Writing Function Definition

In its simplest form, a function is defined in Haskell by: (1) writing the function name, (2) followed by its arguments, (3) then a “=”, (4) followed by the body of the function definition. Such a definition is called an *equation* since it obeys the rules for mathematical equations.

Here are some simple function definitions. There are no parentheses around the arguments; they are simply written in order, and parentheses are used only to avoid ambiguity, as described under the binding rules in Section 7.2.3.1 (page 384), or to improve readability.

```
offset = (ord 'a') - (ord 'A')           -- 0-ary function
suc x   = x+1                             -- successor of x
imply p q = not p || q                    -- boolean implication
isdigit c = ('0' <= c) && (c <= '9')      -- is c a digit?
isupper c = ('A' <= c) && (c <= 'Z')      -- is c uppercase?
islower c = ('a' <= c) && (c <= 'z')      -- is c lowercase?

-- convert uppercase character c to lowercase
uplow c = chr(offset + ord(c))

-- convert lowercase character c to uppercase
lowup c = chr(-offset + ord(c))
```

Variable `offset`, defined above, is actually a function over zero arguments, a 0-ary function (see Section 2.2, page 17). Its value is constant.

7.2.3.3 Multiple Equations in Function Definition

A function may have multiple equations in its definition:

```
imply False q = True
imply True  q = q
```

or, even

```

imply False False = True
imply False True  = True
imply True  False = False
imply True  True  = True

```

To evaluate this function given its argument values, proceed sequentially from top to bottom until the arguments match the parameters in the definition (see pattern matching in Section 7.2.7, page 397).

7.2.3.4 Conditional Equation

In imperative programming, `if-then-else` construct is used to test a predicate and perform calculations based on the result of the test. Haskell also provides an `if-then-else` construct with the expected meaning:

```
absolute x = if x >= 0 then x else (-x)
```

defines the function that computes the absolute value of its argument.

It is often easier to use a *conditional equation* instead of `if-then-else`, as shown below.

```

absolute x
| x >= 0 = x
| x < 0  = -x

```

The conditional equation, above, consists of two *clauses*. A clause starts with a vertical bar (`|`), followed by a predicate (called a *guard*), an equal sign (`=`) and the expression denoting the function value for this case. The guards are evaluated in the order in which they are written (from top to bottom), and for the first guard that is true, its corresponding expression is evaluated and its value is taken as the function value. In the example of the `absolute` function, the guards are mutually exclusive. But they do not need to be. The second guard could be `x <= 0`; when `x = 0`, both guards are true but the evaluation succeeds with the first guard and the corresponding expression value for `x`, which is 0, is returned. There is a runtime error if no guard is true in the evaluation of a conditional equation.

The condition *otherwise*, which can appear only as the last guard, denotes a predicate that holds when none of the other guards hold. The same effect is achieved by writing `True` for the guard in the last equation.

The function `chCase`, below, converts the argument character from upper to lowercase, or lower to uppercase, as is appropriate. We have already seen two functions, `uplow` (to convert from upper to lowercase), `lowup` (to convert from lower

to uppercase), and a function `isupper` whose value is `True` iff its argument is an uppercase letter.

```
chCase c          -- change case
| isupper c = uplow c
| otherwise = lowup c
```

A conditional equation is treated as a single equation even though it may have multiple clauses. A definition may include multiple equations, some of which may be conditional.

7.2.3.5 Functions Defined Using `where` Clause

Consider the function:

```
pythagoras x y z = (x*x) + (y*y) == (z*z)
```

It may be simpler to read it in the following form:

```
pythagoras x y z = sqx + sqy == sqz
      where
        sqx = x*x
        sqy = y*y
        sqz = z*z
```

The `where` construct permits *local definitions*, that is, definitions of functions (and constants, which are 0-ary functions) *within* another function definition. Variables `sqx`, `sqy` and `sqz` are undefined outside this definition.

We can also do this example by using a *local function* `sq` to define squaring.

```
pythagoras x y z = sq x + sq y == sq z
      where sq p = p*p
```

Personal disagreement with Haskell's choice of symbols Designers of Haskell have chosen to use `=` in function definition, so they had to choose a different symbol, `==`, for comparison in arithmetic expressions. I would have preferred using `=` for comparison, as it has been the norm for many centuries, and a different symbol, possibly `:=`, for definition. I prefer $\triangleq$ in a definition in printed texts.

7.2.4 Program Layout

Haskell uses line indentations in the program to delineate the scope of definitions. A definition starts with the name of the entity being defined and ended by any text that begins to the left (column-wise) of the entity name. In Figure 7.1, the left program fragment is properly indented whereas the same program in the right has an indentation error.

<pre>chCase c upper c = uplow c otherwise = lowup c</pre>		<pre>ch1Case c upper c = uplow c otherwise = lowup c</pre>
--	--	---

Figure 7.1 Program layout.

7.2.5 Recursively Defined Functions

Functions may be defined recursively, that is, a function definition may include the name of the function in its defining equations. There is no special syntax to specify recursion in Haskell; in fact, the reader is encouraged to treat recursive definitions exactly as the non-recursive ones. I show a few small examples in this section though recursion is ubiquitous in the rest of this chapter.

7.2.5.1 Length of a List

One of the simplest applications of recursion is in computing the length of a given list:

```
len [ ]      = 0
len (x:xs) = 1 + len xs
```

7.2.5.2 Computing Fibonacci Numbers

Let $\text{fib } n$ be F_n , the n^{th} number in the Fibonacci sequence (see Section 3.2.1.3, page 88). Figure 7.2 shows two different ways of defining fib .

<pre>fib 0 = 0 fib 1 = 1 fib n = fib(n-1) + fib(n-2)</pre>		<pre>fib n n == 0 = 0 n == 1 = 1 True = fib(n-1) + fib(n-2)</pre>
--	--	--

Figure 7.2 Fibonacci function.

Neither program is very efficient because there is considerable amount of recomputation. For example, in computing $(\text{fib } 6)$, both $(\text{fib } 4)$ and $(\text{fib } 5)$ have to be computed, and in computing $(\text{fib } 5)$, $(\text{fib } 4)$ has to be recomputed. I show a better solution in the next example.

7.2.5.3 Computing Pairs of Fibonacci Numbers

The value of function fibpair for argument n is the pair (F_n, F_{n+1}) , the n^{th} and $(n+1)^{\text{th}}$ numbers in the Fibonacci sequence.

```
fibpair 0 = (0,1)
fibpair n = (y, x+y)
  where (x,y) = fibpair (n-1)
```

7.2.5.4 Multiplication of Two Numbers

Consider multiplying natural numbers x and y using addition as the only operation. It is easy to define a simple function:

```
mlt x 0 = 0
mlt x y = (mlt x (y-1)) + x
```

The algorithm has a running time proportional to the magnitude of y , far too inefficient to be of much use. For a better algorithm, observe the following properties of product:

$$\begin{aligned}x \times (2 \times t) &= 2 \times (x \times t) \\x \times (2 \times t + 1) &= 2 \times (x \times t) + x\end{aligned}$$

Function `quickMlt` for multiplication is based on these identities using z for $(x \times t)$. Observe that $2 * z$ is easy to compute by left-shifting z , and $y \text{ `div` } 2$ is the right-shift of y .

```
quickMlt x 0 = 0
quickMlt x y
| even y = 2*z
| odd  y = 2*z+x
  where z = quickMlt x (y `div` 2)
```

Each recursive call is made with a smaller value of the second argument; so, termination is guaranteed. Further, the algorithm runs in $\mathcal{O}(\log y)$ time because y is being nearly halved in each call.

This example illustrates one of the main advantages of recursive over imperative programming style. Function `quickMlt` was derived solely from a few facts about the properties of the multiplication operator. Contrast this with the same program written in imperative style in Figure 4.3 (page 162). Its proof requires a loop invariant that is not obvious.

7.2.5.5 Greatest Common Divisor

The greatest common divisor (gcd) of two positive integers is the largest positive integer that divides both m and n . Euclid gave an algorithm for computing gcd about 2,300 years ago, an algorithm that is still used today.

```
egcd m n
| m == 0 = n
| otherwise = egcd (n `mod` m) m
```

A simpler version of this algorithm, though not as efficient, replaces computing remainders by repeated subtraction:

```
gcd m n
| m == n = m
| m > n  = gcd (m - n) n
| n > m  = gcd m (n - m)
```

Binary greatest common divisor There is a modern version of the gcd algorithm, known as *binary gcd*. This algorithm uses multiplication and division by 2, which are implemented by shifts on binary numbers. The algorithm uses the following facts about gcd m n :

1. if m and n are equal, then gcd m n = m ,
2. if m and n are both even, say $2*s$ and $2*t$, then gcd m n = $2*(gcd\ s\ t)$,
3. if exactly one of m and n , say m , is even and equal to $2*s$, then gcd m n = gcd s n ,
4. if m and n are both odd and, say $m > n$, then gcd m n = gcd $(m-n)$ n . In this case, $m-n$ is even whereas n is odd; so, gcd $(m-n)$ n = gcd $((m-n) \div 2)$ n .

```
bgcd m n
| m == n = m
| (even m) && (even n) = 2 * (bgcd s t)
| (even m) && (odd n) = bgcd s n
| (odd m) && (even n) = bgcd m t
| m > n = bgcd ((m-n) `div` 2) n
| n > m = bgcd m ((n-m) `div` 2)
      where
          s = m `div` 2
          t = n `div` 2
```

To estimate the running time (the number recursive calls) as a function of m and n , note that the value of $\log_2 m + \log_2 n$ decreases in each recursive call. So the execution time is at most logarithmic in the argument values.

7.2.5.6 Writing a Sequence of Function Definitions

A set of functions may be defined in any order in a program. The bottom-up rule is to define each function before it is called. The top-down rule defines them in the reverse order. There is no right way; it is mostly a matter of taste. The top-down rule is obeyed for local functions, defined using *where* clauses. Neither rule can be obeyed for mutually recursive functions that call each other (see Section 7.2.10, page 404).

7.2.5.7 Primitive versus General Recursion

Tail recursion Consider the recursively defined functions of the last section. They all share a common pattern, that the recursive call is made exactly once in an equation as its last “action”. This form of recursion is called *tail recursion*. Tail recursion can be implemented directly as iteration in an imperative language. In fact, most implementations of functional languages implement tail recursion by iteration, thereby avoiding the use of a stack.

Many recursive programs are not tail recursive. For example, `mergesort` (shown in Section 7.2.8.3, page 400) sorts a list of items by sorting its two halves separately and then merging them. Most functions defined on a tree data structure (see Section 7.5, page 424) are not tail recursive.

Primitive and general recursion A *primitive recursive* function has a bound on the depth of every recursive call that can be expressed using elementary number-theoretic functions. Almost all functions, actually all functions, we encounter in practice are primitive recursive. Every function in this chapter (excepting the one below to illustrate non-primitive recursion) are primitive recursive.

Not all computable functions are primitive recursive. Ackermann function (given in Section 3.5.1.5, page 120) is a well-known example. I repeat its mathematical definition below and show a Haskell definition that is identical to the mathematical definition except for syntactic differences.

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

The Haskell definition:

```
ack 0 n = n + 1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n - 1))
```

This function grows extraordinarily fast. For example, $A(4, 2)$ is around $2^{2^{2^{2^2}}}$, or 2^{65536} , in value.

7.2.6 Type

An outstanding feature of Haskell is *static strong typing*. Static typing means that every function in Haskell has a *type* that is known before the program is run. Strong typing means that the compiler enforces the types so that a function is applied only if the arguments are of the required type. Unlike in C++, $3 + (5 \geq 2)$ is not a valid expression; Haskell does not specify how to add an integer to a boolean.

The type of a variable may be specified by the programmer or deduced by the Haskell interpreter. For the expression `3+4`, the interpreter can deduce the type of the operands and the computed value to be integer (not quite, as we will see). For functions defined as below:

```

  imply p q = not p || q
  isdigit c = ('0' <= c) && (c <= '9')

```

the interpreter can figure out that `p` and `q` in the first line are booleans (because `||` is applied only to booleans) and the result is also a boolean. In the second line, it deduces that `c` is a character because of the two comparisons in the right side involving character constants `'0'` and `'9'`, and that the function value is boolean, from the types of the operands.

It is a major achievement of Haskell to determine the types of (almost) all functions statically.

7.2.6.1 Type Expression

Analogous to expressions that denote values, there are *type expressions* that denote the *types* of expressions. A type expression for a basic type, `Int`, `Bool`, `Char` or `String`, is just the name of the type, as shown below (typing `:t e` yields the type of expression `e` in `ghci`):

```

*Main> :t ('0' <= '9')
'0' <= '9' :: Bool
*Main> :t 'c'
'c' :: Char
*Main> :t "Socrates"
"Socrates" :: [Char]
*Main> :t 3+4
3+4 :: Num p => p

```

A `String` like “Socrates” is a list of characters. The type of `3+4` is not `Int` as one would expect; it is more general as explained later under type classes (see Section 7.2.6.3, page 394). For the moment, read `Num p => p` to mean that the type is `p` where `p` is a `Num`, `Num` being an abbreviation for `Number`.

A structured type, tuple or list, is shown as a tuple or list of its component types.

```

*Main> teacher = ("Socrates","Plato")
*Main> course = ("CompSci", 337)
*Main> :t teacher
teacher :: ([Char], [Char])
*Main> :t course
course :: Num b => ([Char], b)
*Main> :t (teacher,course)
(teacher,course) :: Num b => ([Char], [Char]), ([Char], b)

```

Each function has a type, namely, the types of its arguments in order followed by the type of the result, all separated by `->`.

```
*Main> isdigit c = ('0' <= c) && (c <= '9')
*Main> :t isdigit
isdigit :: Char -> Bool
*Main> imply p q = not p || q
*Main> :t imply
imply :: Bool -> Bool -> Bool
```

The type of `isdigit` is a function of one argument of type `Char` whose value has type `Bool`. The type of `imply`, `Bool -> Bool -> Bool`, should be read as `Bool -> (Bool -> Bool)`. That is, `->` is right associative. The type expression says that `imply` is a function of one `Bool` argument whose value is a function; the latter function has one `Bool` argument and a `Bool` value. The type expression for `imply` is explained under “currying” in Section 7.2.9.1 (page 400).

A function is a data object much like an integer. It has a type, and it can be an argument of another function (see Section 7.2.9, page 400). This feature is often described as “functions are first-class objects”.

Haskell compilers use a sophisticated strategy to deduce the type of all expressions and functions so the programmer does not have to declare types. It succeeds in almost all cases, so it catches type violation as in `3 + (5 >= 2)`.

Note on name capitalization Type names (e.g., `Int`, `Bool`) are always capitalized. A function or variable name is never capitalized.

7.2.6.2 Polymorphism

Haskell permits writing functions whose arguments can be *any* type, or any type that satisfies some constraint. The term *Polymorphism* means that a function can accept and produce data of many different types. This feature allows us to define a single sorting function, for example, that can be applied in a very general fashion. Consider the identity function:

```
*Main> identity x = x
*Main> :t identity
identity :: p -> p
```

That is, for *any* type `p`, `identity` accepts an argument of type `p` and returns a value of type `p`.

The type of function `len` that computes the length of a list is `[a] -> Int`. Here `[a]` denotes that `len` accepts a list of *any* type as its argument.

A less trivial example is a function whose argument is a pair and whose value is the same pair with its components exchanged.

```
*Main> exch (x,y) = (y,x)
*Main> :t exch
exch :: (b, a) -> (a, b)
```

Here, *a* and *b* are arbitrary types, not necessarily equal. So, the following are all valid expressions: `exch (3,5)`, `exch (3,"cat")`, `exch ((2,'a'),5)`. The interpreter chooses the most general type for a function so that the largest set of arguments would be accepted.

Type of a list Haskell rigorously enforces the rule that all elements of a list have the same type. This is an essential requirement for a static strong type system. As we have seen, the type of a list whose elements are of type *p* is `[p]`. A very special case is an empty list, one having no items. I write it as `[]`. It appears a great deal in programming. What is the type of `[]`?

```
*Main> :t []
[] :: [a]
```

This says that `[]` is a list of *a*, where *a* is *any* type. Therefore, `[]` can replace an arbitrary list in any expression without causing type violation.

7.2.6.3 Type Classes

Type classes generalize the notion of types. Consider a function whose argument is a pair and whose value is `True` iff the components of the pair are equal.

```
eqpair (x,y) = x == y
```

It is obvious that `eqpair (3,5)` makes sense but not `eqpair (3,'j')`. We would expect the type of `eqpair` to be `(a,a) -> Bool`, but it is more subtle.

```
*Main> :t eqpair
eqpair :: Eq a => (a, a) -> Bool
```

This says that the type of `eqpair` is `(a, a) -> Bool`, for any type *a* that *belongs* to the type class `Eq`, where `Eq` is a *class* that includes only types on which `==` is defined. If *a* is not in `Eq` class, the test `==` in `eqpair` is not defined. Equality is not necessarily defined on all types; in particular, it is not defined for functions.

A type class is a collection of types, each of which has a certain function (or set of functions) defined on it. The `Eq` class contains the types on which `==` is defined. The Haskell compiler specifies how `==` is to be computed for predefined types, and users may define their own types and specify which function, if any, implements `==`. We have already seen the `Num` class; it consists of all types on which typical arithmetic operations (`+`, `*`, etc.) are defined.

Another important type class is `Ord`, types on which a total order is defined over the elements. As an example of its use, consider the following function that sorts a pair of items.

```
sort2 (x,y)
  | x <= y = (x,y)
  | x > y = (y,x)
*Main> :t sort2
sort2 :: Ord a => (a, a) -> (a, a)
```

The type of `sort2` tells us that the function accepts any pair of elements of the same type, provided the type belongs to the `Ord` type class. And `sort2` returns a pair of the same type as its arguments. An order relation is defined by default over most of the basic types. So we can sort pairs of elements of a variety of types, as shown below.

```
*Main> sort2 (5,2)
(2,5)
*Main> sort2 ('g','j')
('g','j')
*Main> sort2 ("Socrates", "Plato")
("Plato","Socrates")
*Main> sort2 (True, False)
(False,True)
*Main> sort2 ((5,3),(5,4))
((5,3),(5,4))
*Main> sort2 (["Euler","Newton","Turing"],["Euclid", "Plato"])
(["Euclid","Plato"],["Euler","Newton","Turing"])
```

The last two examples use lexicographic order for sorting pairs and pairs of lists. In the last example, the result lists are ordered lexicographically in the order shown because string "Euclid" is smaller than "Euler".

7.2.6.4 User-defined Types

A user may define a type. For example, a point in Cartesian coordinates may be defined:

```
data CartPoint = Coord Float Float
```

This declaration introduces two new names, `CartPoint` and `Coord`, a *type* and a *constructor*, respectively. The constructor `Coord` is a function; given two floating point

numbers it creates an instance of type `CartPoint`. Note that `Coord` is capitalized, though defined function names are never capitalized. The type of `Coord` is

```
Coord :: Float -> Float -> CartPoint.
```

A point can alternatively be represented by its polar coordinates, using its distance from origin (radius) and the angle:

```
data PolarPoint = RadAng Float Float
```

The type of `RadAng` is `RadAng :: Float -> Float -> PolarPoint`. The following functions convert points between the coordinate systems.

```
cart_to_polar (Coord x y) =
  RadAng (sqrt (x^2 + y^2)) (atan(y/x))
polar_to_cart (RadAng r theta) =
  Coord (r*cos(theta)) (r*sin(theta))
```

```
*Main> :t cart_to_polar
cartpolar :: CartPoint -> PolarPoint
*Main> :t polar_to_cart
polarcart :: PolarPoint -> CartPoint
```

A user-defined type can also include polymorphic type parameters. Below, `Pair` is a tuple of two components, each of arbitrary type.

```
Prelude> data Pair a b = Tuple a b
Prelude> :t Tuple
Tuple :: a -> b -> Pair a b
```

Type class for user-defined types Types `Cartpoint`, `PolarPoint` and `Pair` have the same status as any other type. We can define equality over these types as follows. Note particularly the declaration for `Pair`; it specifies the constraint that pairs are equal only if the components are equal, so the components must belong to the `Eq` class.

```
instance Eq CartPoint where
  (Coord x y) == (Coord u v) = (x == u && y == v)
instance Eq PolarPoint where
  (RadAng x y) == (RadAng u v) = (x == u && y == v)
instance (Eq a, Eq b) => Eq (Pair a b) where
  (Tuple x y) == (Tuple u v) = (x,y) == (u,v)
```

In order to display or output the value of a user-defined type, say a `CartPoint` with coordinates `(3.0, 2.0)`, the system has to be told how to display a point. The type class `Show` includes types whose element values can be displayed. Such types have an associated function `show` that converts each element value to a string that

is then displayed. Basic data types and structures over them have pre-defined `show` functions, so `show 3` displays `"3"`.

```
Prelude> :t show
show :: Show a => a -> String
```

For user-defined types, such as `CartPoint` and `PolarPoint`, we have to specify their `show` functions. Below, `CartPoint` is displayed as a pair of coordinates in the standard manner. And, `PolarPoint` is displayed in a more verbose style. For `Pair`, we need to also specify the constraint that both component types are in `Show`. Below, `++` denotes string concatenation.

```
instance Show CartPoint where
  show(Coord x y) = show(x,y)
instance Show PolarPoint where
  show(RadAng r theta) =
    "(Radius: " ++ (show r) ++ ", radian: " ++ (show theta) ++ ")"
instance (Show a, Show b) => Show (Pair a b) where
  show(Tuple a b) = "(fst = " ++ show a ++ " , snd = " ++ show b ++ ")"
```

```
Prelude> Coord 3.0 5.0
(3.0,5.0)
Prelude> RadAng 5.0 0.62
(Radius: 5.0, radian: 0.62)
Prelude> Tuple "John" 345
(fst = "John" , snd = 345)
```

Deriving type classes For a user-defined type like `Cartpoint`, its definition of equality is standard in that the corresponding components are equal. In such cases, Haskell permits a declaration of the form `deriving (Eq)` along with type declaration to adopt a standard meaning of equality. Several type classes can be mentioned in the deriving clause.

```
data CartPoint = Coord Float Float
               deriving (Eq,Ord,Show)
```

For ordering of multiple values, Haskell employs lexicographic ordering. And, `show` merely converts data to a string.

```
Prelude> Coord 3.0 5.0 > Coord 2.0 6.0
True
Prelude> show(Coord 3.0 5.0)
"Coord 3.0 5.0"
```

7.2.7 Pattern Matching

Pattern matching is invaluable in function definitions. I introduce the idea using a small example. The function `eq3`, below, determines if the first three items of an argument list `xs` are equal; assume that `xs` has at least three items.

```
eq3 xs = (x1 == x2) && (x2 == x3)
  where
    x1 = head(xs)
    x2 = head(tail(xs))
    x3 = head(tail(tail(xs)))
```

The actual logic of `eq3` is simple, but most of its code goes into extracting the first three items of `xs` and binding them to the variables `x1`, `x2` and `x3`. Pattern matching makes these steps much simpler, as shown below.

```
eq3 (x1:x2:x3:rest) = (x1 == x2) && (x2 == x3)
```

This definition tells the Haskell interpreter that `eq3` is a function over a list that has at least three items. The first three items in order are bound to the names `x1`, `x2` and `x3`, and the remaining part of the list, possibly empty, is named `rest`.

We can say that “`:`” *constructs* a list and pattern matching *deconstructs* the list⁴. Deconstruction can be applied to any data structure, not just lists.

7.2.7.1 Pattern Matching Rules

The expression `(x1:x2:x3:rest)` shown earlier is called a *pattern*. Pattern matching not only prescribes the type and form of its arguments, such as a list above, but also the bindings of variables to parts of the argument data structure by deconstructing the list.

We have already seen simple patterns in the definition of `imply` in Section 7.2.3.3 (page 385) where the patterns are just constants. In general, a pattern, or pattern expression, is a data structure that names constants and variables. The only restriction is: variables are never repeated in a pattern. So, `(x, x)` is not a pattern. A pattern matches an argument if the variables in the pattern can be bound to values such that pattern and the argument match.

A function definition may have multiple equations each of which may have a pattern in its argument part. The sequence of arguments are matched with the patterns of the equations in order from top to bottom. Whenever there is a pattern match, variables in the pattern are bound to values in the arguments, and then the right side of the corresponding equation is evaluated.

7.2.7.2 Wildcard

Consider the definition of `eq3` again in which `(x1:x2:x3:rest)` is the pattern. Variable `rest` never appears in the function definition. Use `_`, the underscore character, instead of `rest` in the definition to indicate that the value bound to `_` is never used. The pattern `_` matches any value. So, the definition of `eq3` could be written as:

```
eq3 (x1:x2:x3:_) = (x1 == x2) && (x2 == x3)
```

4. Not to be confused with the term deconstruction in literary criticism.

7.2.8 Examples of Pattern Matching Over Lists

7.2.8.1 Elementary Functions Over Lists

Functions that are defined over lists typically deconstruct the list using pattern matching. For tail-recursive function definitions, there are two cases to consider, the list is either empty or non-empty. For an empty list, specify the function value. For a non-empty list, bind the head and tail to variables and make a recursive call over the tail. Two examples are shown in Figure 7.3, where `suml` computes the sum and `prodl` the product of the list elements.

<code>suml [] = 0</code>		<code>prodl [] = 1</code>
<code>suml (x:xs) = x + suml xs</code>		<code>prodl (x:xs) = x * prodl xs</code>

Figure 7.3 Elementary functions over lists.

Below, function `cat` has two arguments, an element `e` and a list of lists `xss`. It transforms `xss` by adding `e` as the first element of each list in `xss`. So,

`cat '0' ["abc", "10"]` is `["0abc", "010"]`, and
`cat 3 [[1,2], []]` is `[[3,1,2], [3]]`.

Define `cat` as follows.

```
cat _ [ ] = [ ]
cat e (xs:xss) = (e:xs):(cat e xss)
```

Function `split`, below, partitions the elements of the argument list into two lists, elements with even index go in the first list and with odd index in the second list (list elements are indexed starting at 0). So, `split [1,2,3]` is `([1,3], [2])`.

```
split [ ] = ([ ], [ ])
split (x: xs) = (x:ys, zs)
               where (zs,ys) = split xs
```

7.2.8.2 List Concatenation and Flattening

Function `conc` concatenates two lists of the same type in order; therefore, `conc [1,2] [3,4]` is `[1,2,3,4]` and `conc [1,2] []` is `[1,2]`.

```
conc [ ] ys = ys
conc (x:xs) ys = x : (conc xs ys)
```

There is a built-in operator, `++`, that is the infix version of `conc`; that is, `conc xs ys` is the same as `xs ++ ys`; we have already seen its use in Section 7.2.6.4. The execution time of the given function is proportional to the length of the first argument list, a result that can be proved by a simple induction on the length of the first list.

Function `flatten` takes a list of lists such as `[ [1,2,3], [10,20], [], [30] ]` and flattens it out by putting all the elements into a single list, `[1,2,3,10,20,30]`. Below, `xs` is a list and `xss` is a list of lists.

```
flatten [ ] = [ ]
flatten (xs : xss) = xs ++ (flatten xss)
```

7.2.8.3 Merge Sort

Merge sort is applied to a list of items to sort them in order. Merging two sorted lists to form a single sorted list is quite easy. This is the main part of the algorithm. Observe that duplicate items in the lists are retained, as they should be in a sort function.

```
merge xs [ ] = xs
merge [ ] ys = ys
merge (x:xs) (y:ys)
  | x <= y   = x : (merge xs (y:ys))
  | x > y    = y : (merge (x:xs) ys)
```

The mergesort algorithm divides the input list into two lists, say using `split` of Section 7.2.8.1 (page 399), sorts them recursively, and then merges them to form the desired list.

```
mergesort [ ] = [ ]
mergesort [x] = [x]
mergesort xs = merge (mergesort xs1) (mergesort xsr)
               where (xs1,xsr) = split xs
```

I showed that the running time of mergesort is $\mathcal{O}(n \log n)$ in Section 4.10.1.2 (page 184). The additional space required is $\mathcal{O}(n)$.

7.2.9 Higher Order Functions

A higher order function⁵ has function(s) as arguments. Higher order functions are no different from the functions we have seen so far. They allow a wider latitude in programming, as in defining a function for integration that accepts the function to be integrated as an argument.

7.2.9.1 Currying

Consider function

```
imply p q = not p || q
```

given earlier and another function

```
imply2(x,y)= if x then y else True.
```

5. Higher order functions are sometimes called *functionals*.

Both functions compute the same value given their arguments in proper form. So, are `imply2` and `imply` the same function? The following script shows that they are not because they have different types.

```
*Main> :t imply
imply :: Bool -> Bool -> Bool
*Main> :t imply2
imply2 :: (Bool, Bool) -> Bool
```

We see that `imply2` is a function from a pair of `Bool` to `Bool` whereas `imply`, whose type is `Bool -> (Bool -> Bool)` from the right associativity convention, is a function from `Bool` to a function; the latter function is from `Bool` to `Bool`. That is, `imply` takes its first argument's value and creates a function, call it `temp`. Function `temp` encodes the value of the first argument so that when presented with the second argument it computes the appropriate function value. Another way to interpret `imply` is that its definition is an abbreviation for:

```
imply p q = temp q
  where temp z = not p || z
```

Even though we don't have access to the name of the intermediate function, `temp`, we can still inquire about the type of `imply`.

```
*Main> :t imply
imply :: Bool -> Bool -> Bool
*Main> :t imply True
imply True :: Bool -> Bool
```

The technique of converting a function over multiple arguments to a series of 1-argument functions is known as currying.⁶ There are major theoretical implications of this construction including a correspondence between proofs and programs that I do not pursue here. Our interest here is treating a function as data, much like an integer or boolean, so that it can be used as arguments to other functions.

7.2.9.2 Function `foldr`

Consider the functions in Figure 7.3, `sum1` and `prod1`, that return the sum and product of the elements of the argument lists, respectively. I reproduce the definition in Figure 7.4 for easy reference. Both functions have the same structure except for their unit elements, 0 and 1, and the use of operators `+` and `*`.

I define a higher order function, `foldr`, that has a prefix operator `f`, such as `(+)` or `(*)`, the unit element `u`, such as 0 or 1, and a list of items as arguments. The value

6. Haskell is named after Haskell Curry, the inventor of currying.

<code>suml [] = 0</code>	<code>prodl [] = 1</code>
<code>suml (x:xs) = x + suml xs</code>	<code>prodl (x:xs) = x * prodl xs</code>

Figure 7.4 Similar looking functions.

returned by `foldr` is the value computed by applying the corresponding function, such as `suml` or `prodl`, to the list of items.

```
foldr f u [ ] = u
foldr f u (x:xs) = f x (foldr f u xs)
```

Thus, `foldr (+) 0 [3,5,8]` is `3 + (5 + (8 + 0))`, which is 16. Similarly, `foldr (*) 1 [3,5,8]` is `3 * (5 * (8 * 1))`, which is 120.

For a list of type `[a]`, we would expect the type of `foldr` to be

(type of `f`) $\rightarrow$ (type of `u`) $\rightarrow$ (type of `xs`) $\rightarrow$ (type of result), that is,
`(a  $\rightarrow$  a  $\rightarrow$  a)  $\rightarrow$  a  $\rightarrow$  [a]  $\rightarrow$  a`

The interpreter gives a more general type:

```
*Main> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Function `foldr` belongs to type class `Foldable` that includes lists among its types; I will not explain `Foldable`, just read `t a` as `[a]`. Observe that the two arguments of `f` need not be of the same type; I will show an example of the use of this feature later.

We can now define `suml` and `prodl` along with a number of other functions that operate similarly on a list. Below, infix operators, such as `+` have been converted to their prefix forms, `(+)`. Function `xorl` computes exclusive or over a list of booleans and `maxl` the maximum over a list of natural numbers. Function `flatten` has been shown in Section 7.2.8.2 (page 399).

```
suml    xs = foldr (+) 0      xs
prodl   xs = foldr (*) 1      xs
andl    xs = foldr (&&) True   xs
orl     xs = foldr (||) False xs
xorl    xs = foldr (/=) False xs
maxl    xs = foldr max (-1)   xs
flatten xs = foldr (++) [ ]   xs
```

In fact, each function can be defined without mentioning the list `xs` to which it is being applied, as in: `suml = foldr (+) 0`.

We saw that the two arguments of `f` in `foldr` need not be of the same type. Below, `evenl` determines if all integers of a given list are even integers. For its definition, use function `ev` that has integer `x` and boolean `b` as arguments and returns `(even x) && b`, a boolean, as value.

```
ev x b = (even x) && b
evenl xs = foldr ev True xs
```

So, `evenl [10,20]` is `True` and `evenl [1,2]` is `False`.

Function `foldr` folds the elements of the list starting at the right end, that is, `foldr (+) 0 [3,5,8]` is `3 + (5 + (8 + 0))`. There are other versions that fold from left or work on lists of non-zero length so that the unit element does not have to be specified. The latter feature is needed to compute the maximum or minimum over non-empty lists.

7.2.9.3 Function `map`

Higher order function `map` has two arguments, a function `f` and a list `xs`. The result is a list obtained by applying `f` to each element of `xs`.

```
map f [ ] = [ ]
map f (x:xs) = (f x): (map f xs)
```

So, `map not [True,False]` is `[False,True]`, `map even [2,5]` is `[True,False]`, and `map chCase "Plato"` is `"pLATO"`.

The type of `map` is

```
map :: (a -> b) -> [a] -> [b].
```

This function is so handy that it is often used to transform a list to a form that can be more easily manipulated. For example, determine if all integers in the given list `xs` are even by: `andl (map even xs)`. Here `(map even xs)` creates a list of booleans of the same length as `xs`, such that an element of this list is `True` iff the corresponding element of `xs` is even. Function `andl`, defined using `foldr` in Section 7.2.9.2 (page 401), then takes the conjunction of the elements of this list.

Earlier, function `cat e xss` was defined to append `e` as the head element of each list in a list of lists. It can now be defined simply as `cat e = map (e:)`.

Matrix transposition The power of `map` is illustrated in this example, transposing a matrix. Suppose a matrix is given as a list of its rows where each row is a list of its elements. It is required to transpose the matrix to obtain a list of its columns. As an example, `transpose [[11,12], [21,22], [31,32]]` should have the value `[[11,21,31], [12,22,32]]`. Assume that each sublist in the argument of `transpose`, a row of the matrix, has the same non-zero length.

In the recursive case, below, `(map head xss)` picks up the head element of each row and appends it as the first column of the matrix. And `tail xss` returns the tail of a non-empty list; therefore, `(map tail xss)` returns the list of the tails of all the rows, that is, all the rows after eliminating the first column of the matrix. So, `transpose (map tail xss)` transposes that portion of the matrix.

```
transpose ([ ]:xs) = [ ]
transpose xss = (map head xss) : transpose (map tail xss)
```

Matrix operations are expensive in Haskell, in general, because random access to data is typically inefficient. Matrix transposition in an imperative language is a straightforward operation that would take considerably less time than the program given above.

7.2.10 Mutual Recursion

All of our examples so far have involved recursion in which a function calls itself. It is easy to extend this concept to a group of functions that call each other. This is useful when the functions perform similar, but not quite the same, tasks. Clearly, there can be no order in which the functions are defined; the entire group of functions are defined as a unit and the individual functions within the group in any order. A very trivial example is to define a pair of functions, `even` and `odd`, with their standard meanings over natural numbers (see Figure 7.5).

<code>even 0 = True</code>		<code>odd 0 = False</code>
<code>even x = odd (x-1)</code>		<code>odd x = even (x-1)</code>

Figure 7.5 Mutually recursive functions.

I illustrate the use of mutual recursion using a more interesting example.

Removing unnecessary white spaces from a string The input is an arbitrary string. It is required to transform the string by (1) removing all white spaces in the beginning and (2) reducing all other blocks of white spaces (consecutive white spaces) to a single white space. Thus, the string (where `-` denotes a white space):

```
---Mary---had--a-little---lamb--
```

is transformed to

```
Mary-had-a-little-lamb-
```

Encode the solution as a finite state transducer This paragraph contains a short explanation of finite state transducers. Finite state transducers are theoretical machine models. They are excellent at modeling solutions to the kind of

problem shown here. They can be easily implemented in any programming language, functional or imperative. I do not cover finite state transducers in this book but show a diagram in Figure 7.6 that explains the solution for this particular problem in fairly intuitive terms.

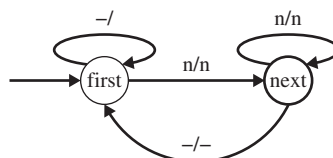


Figure 7.6 Transducer to remove unnecessary white spaces.

The machine has two states, `first` and `next`, and transitions among states shown by the arrows. The initial state is `first`, shown by just an incoming arrow from an unspecified state. The label on an arrow has the form `x/y`, where `x` and `y` are characters, `-` for a white space and `n` for a non-white space. The label `x/y` on an arrow means that processing the next input character `x` in the given state (where the tail of the arrow is incident) causes `y` to be output and a transition to the state given by the head of the arrow. The label `-/` on the arrow incident on `first` means that a white space is processed in the input but there is no corresponding output.

The machine removes all leading white spaces in state `first`, shown by the transition `-/`, and on finding a non-white space outputs the symbol and then transits to state `next`, shown by the transition `n/n`. In state `next`, the machine transforms a string starting with a (1) white space by retaining the white space, and transitioning to `first`, shown by `-/-`, and (2) non-white space by retaining that symbol and remaining in `next`, shown by `n/n`.

Simulate the finite state transducer by a program I design two functions in Figure 7.7, corresponding to the states `first` and `next`, each of which has a list of symbols as its argument. Each transition between states is encoded as a call by one function on another, whereas a transition from a state to itself is just a recursive call on itself. Function `first` is initially called with the given string, in this case `---Mary---had--a-little---lamb--`. The string returned by the combined functions is the desired output.

<code>first []</code>	<code>= []</code>	<code>next []</code>	<code>= []</code>
<code>first('-':ns)</code>	<code>= first(ns)</code>	<code>next('-':ns)</code>	<code>= '-' : first(ns)</code>
<code>first(n:ns)</code>	<code>= n : next(ns)</code>	<code>next(n:ns)</code>	<code>= n : next(ns)</code>

Figure 7.7 Simulating the finite state transducer.

Mutual recursion is the easiest way to encode a finite state machine in a functional language.

7.3 Reasoning About Recursive Programs

We are interested in proving properties of programs, imperative and recursive, for similar reasons: to guarantee correctness, study alternative definitions, and gain insight into the performance of the program. Our main tool in proving properties of functional programs, as in imperative programs, is *induction*. I use the induction principle over natural numbers, described in Sections 3.1.1 (page 85) and 3.1.2 (page 86), well-founded induction in Section 3.5 (page 115), and structural induction in Section 3.6 (page 122).

Proof structure often mimics the program structure. A proof about a function defined over natural numbers will, most likely, use weak or strong induction on numbers. A proof over structured data will use structural induction. For example, a proof about a list that applies tail recursion, say in computing the sum of the numbers in a list, can be proved by appealing to the structure of the list: prove the result for the empty list, and assuming that the result holds for any list xs , prove it for $x:xs$. A proof about a tree would typically prove the result for its subtrees and then deduce it for the whole tree, a structural induction. Certain problems require well-founded induction; I will use it to prove properties of Ackermann function.

In this section I deal only with terminating functions, that is, functions that terminate for every value of their arguments satisfying the preconditions. The proof of termination itself requires induction, that each recursive call is made to a smaller argument according to some well-founded metric.

7.3.1 Fibonacci Numbers, Yet Again

We start with an utterly simple example, the correctness of a function to compute pairs of Fibonacci numbers that was described in Section 7.2.5.3. It is reproduced below.

```
fibpair 0 = (0,1)
fibpair n = (y, x+y)
             where (x,y) = fibpair (n-1)
```

I show that $\text{fibpair}(n) = (F_n, F_{n+1})$, the n^{th} and $(n+1)^{\text{th}}$ number in the Fibonacci sequence. The proof is by weak induction on n .

- $n = 0$: The proof of $\text{fibpair}(0) = (F_0, F_1)$ follows from the first line of the function declaration and the definitions of F_0 and F_1 .

- $n > 0$: Inductively, assume $\text{fibpair}(n-1) = (F_{n-1}, F_n)$. Substitute $(x, y) = (F_{n-1}, F_n)$ in the where clause to deduce $\text{fibpair } n = (y, x + y)$, which is $(F_n, F_{n-1} + F_n) = (F_n, F_{n+1})$, as required.

7.3.2 Number of Non-zero Bits in a Binary Expansion

Function `count` applied to a natural number returns the number of 1s in its binary expansion. The definition of `count`, below, scans the binary expansion of a number from the lower order to the higher order bits, using functions `even` and `odd`. Division by 2 shifts the binary expansion of n to the right by 1 position.

```
count 0 = 0
count n
  | even n = count (n `div` 2)
  | odd  n = count (n `div` 2) + 1
```

Rewrite the definition of `count`, as follows, to simplify mathematical manipulations.

```
count(0) = 0
count(2 × s) = count s
count(2 × s + 1) = (count s) + 1
```

I show that for naturals m and n :

$$\text{count}(m + n) \leq (\text{count } m) + (\text{count } n)$$

The result is immediate if either m or n is zero. But the general result is not obvious. If you consider the addition process for binary numbers, you will be doing some of the steps of the proof given next.

Consider four cases: $(\text{even } m) \wedge (\text{even } n)$, $(\text{even } m) \wedge (\text{odd } n)$, $(\text{odd } m) \wedge (\text{even } n)$, $(\text{odd } m) \wedge (\text{odd } n)$. The second and third cases are identical because m and n are interchangeable due to symmetry in the problem statement. I will write $m = 2 \times s$ for $(\text{even } m)$ and $m = 2 \times s + 1$ for $(\text{odd } m)$, and similarly for n . I show two of the harder proofs; the other proofs are similar. I apply induction over pairs of naturals in all cases. Call a pair (p, q) *smaller* than (m, n) if $p + q < m + n$, which is a well-founded order.

- $\text{count}(m + n) \leq (\text{count } m) + (\text{count } n)$ for $(\text{even } m \wedge \text{odd } n)$:

We have $m = 2 \times s$ and $n = 2 \times t + 1$, for some s and t .

$$\begin{aligned} & \text{count}(m + n) \\ = & \{m = 2 \times s \text{ and } n = 2 \times t + 1\} \\ & \text{count}(2 \times s + 2 \times t + 1) \\ = & \{\text{arithmetic}\} \end{aligned}$$

$$\begin{aligned}
& \text{count}(2 \times (s + t) + 1) \\
= & \{ \text{definition of count, given above} \} \\
& \text{count}(s + t) + 1 \\
\leq & \{ \text{induction hypothesis: } s + t < 2.s + 2.t + 1 = m + n \} \\
& (\text{count } s) + (\text{count } t) + 1 \\
= & \{ \text{definition of count, given above} \} \\
& \text{count}(2 \times s) + \text{count}(2 \times t + 1) \\
= & \{ m = 2 \times s \text{ and } n = 2 \times t + 1 \} \\
& \text{count}(m) + \text{count}(n)
\end{aligned}$$

- $\text{count}(m + n) \leq (\text{count } m) + (\text{count } n)$ for $(\text{odd } m \wedge \text{odd } n)$:

We have $m = 2 \times s + 1$ and $n = 2 \times t + 1$, for some s and t .

$$\begin{aligned}
& \text{count}(m + n) \\
= & \{ m = 2 \times s + 1 \text{ and } n = 2 \times t + 1 \} \\
& \text{count}(2 \times s + 1 + 2 \times t + 1) \\
= & \{ \text{arithmetic} \} \\
& \text{count}(2 \times (s + t + 1)) \\
= & \{ \text{definition of count, given above} \} \\
& \text{count}(s + t + 1) \\
\leq & \{ \text{induction hypothesis: } s + t + 1 < 2.s + 1 + 2.t + 1 = m + n \} \\
& (\text{count } s) + (\text{count } (t + 1)) \\
\leq & \{ \text{induction on the second term: } t + 1 < 2.s + 1 + 2.t + 1 = m + n \} \\
& (\text{count } s) + (\text{count } t) + \text{count}(1) \\
< & \{ \text{count}(1) = 1, \text{ from the definition of count} \} \\
& ((\text{count } s) + 1) + ((\text{count } t) + 1) \\
= & \{ \text{definition of count, given above} \} \\
& \text{count}(2 \times s + 1) + \text{count}(2 \times t + 1) \\
= & \{ m = 2 \times s + 1 \text{ and } n = 2 \times t + 1 \} \\
& \text{count}(m) + \text{count}(n)
\end{aligned}$$

7.3.3 A Reversal and Concatenation Function for Lists

Note: It is more convenient to use typewriter font in this proof.

Given below is the definition of function `rv` that has three arguments, all lists. It concatenates the first list, reversal of the second list and the third list. Its code is:

```

rv [ ] [ ] q = q
rv [ ] (v:vs) q = rv [ ] vs (v:q)
rv (u:us) vs q = u:(rv us vs q)

```

It is easy to see that both list reversal and list concatenation are special cases of `rvc`:

```
reverse ys    = rvc [ ] ys [ ]
xs ++ ys      = rvc xs [ ] ys
```

I prove the following property of `rvc`:

$$\text{rvc } us \text{ } vs \text{ } q = us ++ \text{reverse}(vs) ++ q. \quad (\text{RVC})$$

where `++` is the infix concatenation operator and `reverse` reverses a list.

Facts about `reverse` and `++` The proof of RVC uses the following facts about `reverse` and `++` where `x` is a list item and `xs`, `ys` and `zs` are lists.

- (i) `++` is associative, that is, `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.
- (ii) `[ ]` is a unit of `++`, that is, `[ ] ++ ys = ys` and `ys ++ [ ] = ys`.
- (iii) `x:xs = [x] ++ xs`.
- (iv) `reverse [ ] = [ ]`.
- (v) `reverse [x] = [x]`.
- (vi) `reverse(xs ++ ys) = (reverse ys) ++ (reverse xs)`.

Most of these facts are easily proved. For example, fact (v) follows from (iii), (iv) and (vi). Fact (vi) is the hardest one to prove, its proof follows in the same style as the proof given here for `rvc`.

Proof of RVC The proof uses induction on lists and pairs of lists. For lists `xs` and `ys`, `xs ≤ ys` if `xs` is a suffix of `ys`. And order pairs of lists by lexicographic order. It is possible to use simpler metrics to order pairs of lists, such as the combined lengths of the two lists; I have chosen this metric to illustrate the use of well-founded induction.

In order to prove (RVC), I consider three cases, exactly the ones used in the definition of `rvc`: (1) `us` and `vs` both empty, (2) `us` empty and `vs` non-empty, and (3) `us` non-empty.

- `rvc [ ] [ ] q = [ ] ++ reverse([ ]) ++ q`: From the definition of `rvc`, the left side is `q`. Using facts (iv) and (ii) the right side is `q`.

- `rvc [ ] (v:vs) q = [ ] ++ reverse(v:vs) ++ q`:

$$\begin{aligned} & \text{rvc [] (v:vs) } q \\ = & \text{\{definition of rvc\}} \\ & \text{rvc [] vs (v:q)} \\ = & \text{\{induction hypothesis\}} \end{aligned}$$

$$\begin{aligned}
& [] ++ \text{reverse}(vs) ++ (v:q) \\
= & \{\text{fact (ii)}\} \\
& \text{reverse}(vs) ++ (v:q) \\
= & \{\text{facts (iii) and (i)}\} \\
& \text{reverse}(vs) ++ [v] ++ [q]
\end{aligned}$$

And,

$$\begin{aligned}
& [] ++ \text{reverse}(v:vs) ++ q \\
= & \{\text{fact (ii)}\} \\
& \text{reverse}(v:vs) ++ q \\
= & \{\text{fact(iii)}\} \\
& \text{reverse}([v] ++ [vs]) ++ q \\
= & \{\text{fact(vi)}\} \\
& \text{reverse}(vs) ++ \text{reverse}([v]) ++ q \\
= & \{\text{fact(v)}\} \\
& \text{reverse}(vs) ++ [v] ++ q
\end{aligned}$$

• $\text{rvc } (u:us) \text{ vs } q = (u:us) ++ \text{reverse}(vs) ++ q$:

$$\begin{aligned}
& \text{rvc } (u:us) \text{ vs } q \\
= & \{\text{definition of rvc}\} \\
& u:(\text{rvc } us \text{ vs } q) \\
= & \{\text{induction hypothesis}\} \\
& u:(us ++ \text{reverse}(vs) ++ q) \\
= & \{\text{fact (iii)}\} \\
& [u] ++ us ++ \text{reverse}(vs) ++ q \\
= & \{\text{fact (iii)}\} \\
& (u:us) ++ \text{reverse}(vs) ++ q
\end{aligned}$$

7.3.4 Reasoning About Higher Order Functions

I show a few simple proofs involving higher order functions. All the proofs use Leibniz' law⁷ that two functions f and g are equal if (1) their domains and codomains are equal, and (2) for all argument values the function values are equal, that is, $f(x) = g(x)$ for all x in their domain.

7. Ascribed to Gottfried Wilhelm Leibniz, an outstanding German mathematician of the mid-17th century.

7.3.4.1 A Proof About filter

Function `filter` applies a predicate, a function whose value is boolean, to the items of a list and retains only those items that satisfy the predicate. So, `filter` is a higher order function.

```
filter p [ ] = [ ]
filter p (x:xs)
  | p x      = x: (filter p xs)
  | otherwise = (filter p xs)
```

Thus, `filter even [2,3,4]` is `[2,4]`, `filter isdigit ['a','9','b','0','c']` is `"90"`, and `filter isdigit "Plato"` is `""`. I prove that

$\text{filter } p (\text{filter } p \text{ } xs) = \text{filter } p \text{ } xs$, for all p and xs .

Proof is by induction on the length of xs .

- $\text{filter } p (\text{filter } p []) = \text{filter } p []$:

$$\begin{aligned}
 & \text{filter } p (\text{filter } p []) \\
 = & \quad \{\text{definition of } \text{filter} \text{ applied within parentheses}\} \\
 & \text{filter } p [] \\
 = & \quad \{\text{definition of } \text{filter}\} \\
 & []
 \end{aligned}$$

Thus, both $\text{filter } p (\text{filter } p [])$ and $\text{filter } p []$ are `[]`, hence equal.

Inductive case, $\text{filter } p (\text{filter } p (x : xs)) = \text{filter } p (x : xs)$. Consider two cases, p holds for x and p does not hold for x .

- $\text{filter } p (\text{filter } p (x : xs)) = \text{filter } p (x : xs)$, where p holds for x :

$$\begin{aligned}
 & \text{filter } p (\text{filter } p (x : xs)) \\
 = & \quad \{\text{definition of } \text{filter} \text{ applied to the inner function, } p \text{ holds for } x\} \\
 & \text{filter } p (x : (\text{filter } p \text{ } xs)) \\
 = & \quad \{\text{definition of } \text{filter} \text{ applied to the outer function, } p \text{ holds for } x\} \\
 & x : \text{filter } p (\text{filter } p \text{ } xs) \\
 = & \quad \{\text{induction}\} \\
 & x : (\text{filter } p \text{ } xs) \\
 = & \quad \{\text{definition of } \text{filter}, p \text{ holds for } x\} \\
 & \text{filter } p (x : xs)
 \end{aligned}$$

- $\text{filter } p (\text{filter } p (x : xs)) = \text{filter } p (x : xs)$, where p does not hold for x :

$$\begin{aligned}
 & \text{filter } p (\text{filter } p (x : xs)) \\
 = & \quad \{\text{definition of } \text{filter} \text{ applied to the inner function, } p \text{ does not hold for } x\} \\
 & \text{filter } p (\text{filter } p xs) \\
 = & \quad \{\text{induction}\} \\
 & \text{filter } p xs \\
 = & \quad \{\text{definition of } \text{filter}, p \text{ does not hold for } x\} \\
 & \text{filter } p (x : xs)
 \end{aligned}$$

7.3.4.2 Function Composition is Associative

I show that for functions f , g and h , the identity $(f \circ g) \circ h = f \circ (g \circ h)$ holds, where $\circ$ denotes function composition. Composition $\circ$ is a higher order function; its type, as reported by the Haskell interpreter, is $(t1 \rightarrow t2) \rightarrow (t3 \rightarrow t1) \rightarrow t3 \rightarrow t2$.

The following proof uses Leibniz' law and the definition of function composition, that $(f \circ g)(x) = f(g(x))$. It does not use induction.

- $(f \circ g) \circ h = f \circ (g \circ h)$

$$\begin{aligned}
 & ((f \circ g) \circ h)(x) \\
 = & \quad \{\text{definition of function composition}\} \\
 & (f \circ g)(h(x)) \\
 = & \quad \{\text{definition of function composition}\} \\
 & f(g(h(x)))
 \end{aligned}$$

And,

$$\begin{aligned}
 & (f \circ (g \circ h))(x) \\
 = & \quad \{\text{definition of function composition}\} \\
 & f((g \circ h)(x)) \\
 = & \quad \{\text{definition of function composition}\} \\
 & f(g(h(x)))
 \end{aligned}$$

7.3.4.3 Function `map` Distributes Over Composition

I show that `map` distributes over composition, $(\text{map } f) \circ (\text{map } g) = (\text{map } (f \circ g))$. The proof follows the inductive strategy: show the result for any list xs , by induction on the length of xs :

$$((\text{map } f) \circ (\text{map } g)) xs = (\text{map } (f \circ g)) xs.$$

- $((\text{map } f) \circ (\text{map } g)) [] = (\text{map } (f \circ g)) []$:

$$\begin{aligned}
 & ((\text{map } f) \circ (\text{map } g)) [] \\
 = & \quad \{\text{definition of composition}\} \\
 & (\text{map } f)(\text{map } g []) \\
 = & \quad \{\text{definition of } \text{map}\} \\
 & (\text{map } f) [] \\
 = & \quad \{\text{definition of } \text{map}\} \\
 & []
 \end{aligned}$$

And,

$$\begin{aligned}
 & (\text{map } (f \circ g)) [] \\
 = & \quad \{\text{definition of } \text{map}\} \\
 & []
 \end{aligned}$$

- $((\text{map } f) \circ (\text{map } g))(x : xs) = (\text{map } (f \circ g))(x : xs)$:

$$\begin{aligned}
 & ((\text{map } f) \circ (\text{map } g))(x : xs) \\
 = & \quad \{\text{definition of composition}\} \\
 & (\text{map } f)(\text{map } g(x : xs)) \\
 = & \quad \{\text{definition of } \text{map} \text{ applied to the inner function}\} \\
 & (\text{map } f)((g x) : (\text{map } g xs)) \\
 = & \quad \{\text{definition of } \text{map} \text{ applied to the outer function}\} \\
 & (f(g x)) : (\text{map } f(\text{map } g xs)) \\
 = & \quad \{\text{induction}\} \\
 & (f(g x)) : (\text{map } (f \circ g) xs)
 \end{aligned}$$

And,

$$\begin{aligned}
 & (\text{map } (f \circ g))(x : xs) \\
 = & \quad \{\text{definition of } \text{map}\} \\
 & ((f \circ g) x) : (\text{map } (f \circ g) xs) \\
 = & \quad \{\text{definition of composition applied to the first term}\} \\
 & (f(g x)) : (\text{map } (f \circ g) xs)
 \end{aligned}$$

7.4 Recursive Programming Methodology

I discuss several methodological aspects of recursive programming and illustrate them using examples.

7.4.1 Alternative Recursive Problem Formulations

All of our examples so far show a single way of decomposing a problem for recursive computation. In practice, a problem may be decomposed in a variety of ways. I show alternative decompositions in the problem of evaluating a polynomial.

A polynomial in variable x has the form $a_n \times x^n + a_{n-1} \times x^{n-1} + \dots + a_0 \times x^0$, where n is the degree of the polynomial, each coefficient a_i is a real (or complex) number, and the highest coefficient a_n is non-zero. For example, a polynomial of degree 3 is $2 \times x^3 - 3 \times x^2 + 0 \times x + 4$.

Horner's rule for sequential polynomial evaluation Evaluating a polynomial at a given value of x , say $2 \times x^3 - 3 \times x^2 + 0 \times x + 4$ at $x = 2$, gives us $2 \times 2^3 - 3 \times 2^2 + 0 \times 2 + 4 = 16 - 12 + 0 + 4 = 8$. Write the polynomial as

$$\begin{aligned} & a_0 + a_1 \times x + a_2 \times x^2 + \dots + a_n \times x^n \\ = & a_0 + x \times (a_1 + a_2 \times x + \dots + a_n \times x^{n-1}) \end{aligned}$$

The second term requires evaluation of a polynomial of lower degree, that is, $a_1 + a_2 \times x + \dots + a_n \times x^{n-1}$. We can translate this scheme directly to a recursive program. Below, the polynomial is represented as a list from its least significant to its most significant coefficients, and x is a specific value at which the polynomial is being evaluated. The empty polynomial, that is, one with no terms, evaluates to 0.

```
eps [ ] x = 0
eps (a:as) x = a + x * (eps as x)
```

Function `eps` evaluates $2 \times x^3 - 3 \times x^2 + 0 \times x + 4$ as follows.

$$\begin{aligned} & 4 + 0 \times x - 3 \times x^2 + 2 \times x^3 \\ = & 4 + x \times (0 - 3 \times x + 2 \times x^2) \\ = & 4 + x \times (0 + x \times (-3 + 2 \times x)) \\ = & 4 + x \times (0 + x \times (-3 + x \times (2))) \end{aligned}$$

The function is described in a tail-recursive fashion, so it can be easily implemented by an iterative algorithm.

Parallel polynomial evaluation It is possible to evaluate different parts of a polynomial *simultaneously*. Of course, the Haskell implementation is on a sequential computer, so the evaluation will be done in sequential fashion. However, the form of recursion shows potential for parallel evaluation. This technique is used in the Fast Fourier Transform algorithm (see Section 8.5.3, page 475).

The polynomial is decomposed into two parts, one with coefficients of even indices and the other with those of odd indices.

$$\begin{aligned}
& a_0 + a_1 \times x + a_2 \times x^2 + \dots + \dots \\
= & (a_0 + a_2 \times x^2 + \dots + a_{2i} \times x^{2i} + \dots) \\
& + (a_1 \times x + a_3 \times x^3 + \dots + a_{2i+1} \times x^{2i+1} + \dots) \\
= & (a_0 + a_2 \times x^2 + \dots + a_{2i} \times x^{2i} + \dots) \\
& + x \times (a_1 + a_3 \times x^2 + \dots + a_{2i+1} \times x^{2i} + \dots)
\end{aligned}$$

The value of the polynomial is given in terms of two polynomial values at x^2 , one with the coefficients with even indices and the other with odd indices, and the second term is additionally multiplied by x . Evaluation of each polynomial may be further parallelized recursively. We study parallel recursion in Chapter 8 and polynomial evaluation, in particular, in Section 8.5.2 (page 474).

The division of the input list of coefficients into those with even and odd indices is easy. The function `split`, introduced for merge sort in Section 7.2.8.3 (page 400) serves that purpose.

```

epp [ ] x = 0
epp [c] x = c
epp as x = (epp asl sqx) + x * (epp asr sqx)
      where
        (asl,asr) = split as
        sqx = x*x

```

It is essential to have an equation for a singleton list, `epp [c] x = c`, otherwise the computation will not terminate.

7.4.2 Helper Function

It is rare to find a function definition of any complexity that uses itself as the only function in the recursive definition. Most function definitions require introducing additional functions. These additional functions are called *helper functions*. Often a helper function is defined within the scope of the original function in a `where` clause. Sometimes it is more convenient to define a helper function in an outer scope where it may be shared in the definitions of a number of other functions.

Helper functions are ubiquitous, and their use is extensive in the rest of this chapter. I show a helper function in the next example.

7.4.2.1 Enumerating Permutations

The problem is to enumerate all permutations of a list of distinct items. The required result is a list of lists, where each sublist is a distinct permutation. I show a solution here using two helper functions and a more sophisticated solution in Section 7.4.3.5 (page 421).

We can apply recursion in the obvious fashion. For list $(x:xs)$, recursively compute all permutations of xs . Next, insert x in each possible position in all the permutations of xs . For insertion of x , I define two helper functions.

First, define function `insert` over item x and list xs to insert x in all possible positions within xs . The result is a list of lists. Recall that `map (y:)` appends y as the head of each list in a list of lists.

```
insert x [ ]      = [[x]]
insert x (y:ys) = (x:y:ys) : (map (y:) (insert x ys))

*Main> insert 3 [1,2]
[[3,1,2],[1,3,2],[1,2,3]]
```

Next, generalize `insert` to insert an item in all positions in a list of lists, not just a single list. The result is a list of lists. The definition uses `insert` as a helper function.

```
insertall x [ ]      = [ ]
insertall x (xs:xss) = (insert x xs) ++ (insertall x xss)

*Main> insertall 3 [[1,2],[2,1]]
[[3,1,2],[1,3,2],[1,2,3],[3,2,1],[2,3,1],[2,1,3]]
```

Now, it is simple to define `permute` using `insertall` as a helper function. The argument list of `permute` can be of any type as long as it has distinct elements.

```
permute [ ]      = [ ]
permute [x]      = [[x]]
permute (x:xs) = insertall x (permute xs)

*Main> permute "abc"
["abc","bac","bca","acb","cab","cba"]
```

7.4.3 Function Generalization

It is often helpful to compute a more general function than the one required. Sometimes a more general function may be easier or more efficient to compute. This is reminiscent of generalization of a predicate to construct an inductive hypothesis (see Section 3.3.3, page 95).

I start with a simple example. It is easy to compute the maximum, `topfirst`, of a list of distinct numbers. The recursive step for `topfirst` has the form,

$$\text{topfirst}(x:xs) = \max x (\text{topfirst } xs).$$

Now, suppose we want to compute the *second* largest number, `topsecond`, of a list of distinct numbers that has at least two elements. There is no similar recursive definition; `topsecond(x:xs)` cannot be computed from x and `topsecond xs`. The most

reasonable solution is to compute both the maximum and the second maximum using a more general helper function, `toptwo`, then retain only the relevant value for `topsecond`.

```

toptwo[x,y] = (max x y, min x y)
toptwo(x:xs) = (f,s)
  where
    (f',s') = toptwo(xs)
    f = max x f'
    s = max s' (min x f')

topsecond xs = snd (toptwo xs)

```

I consider two kinds of function generalization in this section: (1) in order to define function f , define a more general function g over the given arguments and show that f is readily computable from g , and (2) define a function on an extended set of arguments and show that the original function is obtained by setting one (or more) of the extended arguments to a specific value. I propose a heuristic below for generalizations of the first kind. There is no general heuristic for the second kind of generalization, though it is related to Theorem 4.1 (page 165) of Chapter 4.

7.4.3.1 Extension Heuristic for Function Generalization Over Lists

I suggest a heuristic, called *Extension heuristic*, for generalizations of functions over lists. Consider the example of computing the second largest number. We could not directly define `topsecond` because `topsecond(x:xs)` is not computable from x and `topsecond xs`. That is, `(topsecond xs)` does not carry enough information to extend the computation to `topsecond(x:xs)`. The suggested heuristic is based on the view that a function condenses the information about its argument, $f(xs)$ is a digest of xs . A recursive definition of f is possible if the digest is sufficient to compute the digest for $(x:xs)$ using x . The function that carries most information is the identity function whose digest is the complete argument sequence.

Function g is an *extension* of f if $f(x:xs)$ is computable from the pair $(x, g(xs))$, for all x and xs . Then it is sufficient to compute g to extend the computation of f . Function f is *inductive* if it is an extension of itself. Then its computation can be extended element by element over a list starting with $f([])$. The complexity of computing $f(x:xs)$ from $(x, g(xs))$ is not the point of this discussion.

Observation 7.1 Given a sequence of functions $\langle f_0, f_1 \dots f_n \rangle$ such that each f_{i+1} is an extension of f_i and f_n is inductive, $f_0(xs)$ for any xs can be incrementally computed, that is, by scanning the elements of xs one at a time.

Proof. Apply induction on k downward from n to 0. Assume that each $f_i([\])$ is computable.

- $f_n(xs)$ is computable for all xs : We are given that $f_n([\])$ is computable and that $f_n(x:xs)$ is computable from $(x, f_n(xs))$ because f_n is inductive. The result follows by induction on the length of xs .
- $f_k(xs)$ is computable given $f_{k+1}(xs)$ is computable, for all xs : The proof is similar to the one above. ■

Observation 7.1 suggests a heuristic for defining f based on successive approximation: starting with f , postulate a sequence of extensions until an extension is inductive. We seek *minimal* extensions at each stage, where, intuitively, g is a minimal extension of f if $g(xs)$ has the smallest amount of additional information beyond $f(xs)$. Clearly, there is no easy way to prove that an extension is minimal, nor is it necessary. The heuristic merely asks for an extension, and it is best to try one that is intuitively as small as possible. The most minimal extension of f is f itself provided f is inductive; the maximal extension is the identity function. This approach is described in Misra [1978] and its theoretical foundation justified in Gupta and Misra [2003].

7.4.3.2 Maximum Segment Sum

A *segment* within a given list of real numbers is a contiguous subsequence, possibly empty, and *segment sum* is the sum of the elements of the segment. We seek a *maximum segment*, a segment having the maximum sum. The problem is interesting only because the numbers in the list could be positive, negative or zero. So, the maximum segment could possibly be empty, whose segment sum is 0. This problem is also treated as the problem of finding a longest path in an acyclic graph in Section 6.9.3.1 (page 327).

Let $mss(xs)$ be the value of maximum segment sum of list xs . I show how to compute $mss(xs)$ and leave it as an exercise to modify the algorithm to return the maximum segment. The development uses the extension heuristic of Section 7.4.3.1.

Observe that mss is not inductive. To see this, consider $mss[-1, 4]$ and $mss[-3, 4]$, both of which have the same value of 4. So, the pairs $(2, mss[-1, 4]) = (2, 4)$ and $(2, mss[-3, 4]) = (2, 4)$ are identical. Yet, $mss[2, -1, 4]$ and $mss[2, -3, 4]$ have different values, 5 and 4, respectively, because their maximum segments are $[2, -1, 4]$ and $[4]$.

In analyzing this counterexample, it is easy to see the source of the problem and how to rectify it. Function value $mss(xs)$ does not carry any information about a segment that *starts* at the beginning of xs but that is not the maximum segment, so a maximum segment of $x:xs$ starting at x cannot be identified.

Let $mss'(xs)$ be a pair of values (c, m) where c is the value of the maximum segment starting at the beginning of xs and m the value of the maximum segment anywhere within xs ; they could be identical values. I claim that mss' is an extension of mss . It meets the condition for extension because $mss(x:xs)$ is computable from $(x, mss'(xs)) = (x, (c, m))$ as follows. The maximum segment of $(x:xs)$ either (1) starts at x and then its value is the larger of 0 (for the empty segment) and $x + c$, or (2) it does not start at x and its value is m . So, $mss(x:xs) = \max(0, x + c, m)$.

The next step is to determine if mss' is inductive, otherwise find a minimal extension for it. It turns out that mss' is inductive, that is, $mss'(x:xs) = (c', m')$ is computable from $(x, mss'(xs)) = (c, m)$. From the discussion in the last paragraph, $c' = \max(0, x + c)$ and $m' = \max(m, c')$. The corresponding program is shown in Figure 7.8.

Maximum segment sum

```

mss' [ ]      = (0,0)
mss' (x:xs)   = (c',m')
               where
                   (c,m) = mss' xs
                   c'  = max 0 (x+c)
                   m'  = max m c'

mss xs = m
       where (_,m) = mss' xs

```

Figure 7.8

The definition of mss' is tail-recursive; so, its implementation can be iterative. Its computation time is linear in the length of the given list, and it takes only a constant amount of extra space. This is an elementary application of dynamic programming (see Section 7.6, page 432).

7.4.3.3 Prefix Sum

The *prefix sum* of a list of numbers is a list of equal length whose i^{th} element is the sum of the first i items of the original list. So, the prefix sum of $[3, 1, 7]$ is $[3, 4, 11]$. The problem is of interest for any associative operator $\oplus$, not just sum, in which case the prefix sum of $[3, 1, 7]$ is $[3, 3 \oplus 4, 3 \oplus 4 \oplus 7]$. As examples, using integer multiplication as the operator, the prefix sum of $[3, 1, 7]$ is $[3, 3, 21]$, and using the

associative operator `++` over strings, the prefix sum of `["The", "quick", "brown"]` is `["The", "The quick", "The quick brown"]`.

This problem arises in a variety of contexts. I consider parallel algorithms for the problem in Section 8.5.5 (page 485) and show its application in the design of an adder circuit.

A straightforward solution for the problem is:

```
ps [ ] = [ ]
ps (x:xs) = x: (map (x+) (ps xs))
```

where `map (x+)ys` adds `x` to every element of list `ys`. The implementation of the function computes the prefix sum over the tail of the list, then adds the head element to each item of the computed list and appends the head. So, the computation time obeys a recurrence of the form $f(n+1) = (n+1) + f(n)$ with a solution of $\mathcal{O}(n^2)$. I propose a linear solution below, similar to what would be computed in an iterative scheme, by introducing an additional argument.

Code function `ps'` of two arguments, a list `xs` and a number `c` so that `(ps' xs c)` is `(ps xs)` with `c` added to each element of the list. Then `ps xs` is `(ps' xs 0)`.

```
ps xs = ps' xs 0
      where ps' [ ] c = [ ]
            ps' (x:xs) c = (c+x) : (ps' xs (c+x))
```

The implementation of `ps'` takes linear time in the length of the list.

7.4.3.4 List Reversal

I show with this example how function generalization can realize a more efficient computation, similar to the prefix sum example of Section 7.4.3.3.

Reversing the order of items in a given list is an important function. There is a standard function, `reverse`, in Haskell library. The naive definition for a similar function `rev` is as follows.

```
rev [ ] = [ ]
rev (x: xs) = (rev xs) ++ [x] -- concatenate x to xs
```

The running time of this algorithm is $\mathcal{O}(n^2)$, where n is the length of the argument list. (Prove this result using a recurrence relation. Use the fact that `++` takes $\mathcal{O}(n)$ time in the length n of its first argument.) In imperative programming, an array can be reversed in linear time. Something seems terribly wrong with functional programming! Actually, we *can* attain a linear computation time in functional programming by using function generalization.

Define function `rev'` over two argument lists `xs` and `ys` so that

```
rev' xs ys == (rev xs) ++ ys.
```

Then `rev' xs [ ] == (rev xs)`.

The definition of `rev'` is:

```
rev' [ ] ys      = ys
rev' (x:xs) ys = rev' xs (x:ys)
```

In each recursive call, the length of the first argument list decreases by 1, so the number of recursive calls is equal to the size of the first argument list, and the computation time is linear in that list size. See the proof of a more general list reversal function in Section 7.3.3 (page 408).

7.4.3.5 Enumerating Permutations

The problem is to enumerate all permutations of a list of distinct items. The required result is a list of lists, where each sublist is a distinct permutation. I showed a solution in Section 7.4.2.1 (page 415) using helper functions. I show a different solution here using function generalization, by introducing an additional argument in the function definition.

The strategy is to designate a particular item, x , as the first item of every permutation. Enumerate, recursively, all permutations of the remaining items. Then add x as the head of every enumerated list. This procedure enumerates all permutations in which x is the head item; so, repeat this procedure treating each item as x .

Define function `permute'` of two arguments, both lists of distinct items, say `xs` and `ys`. The result of `permute' xs ys` is the set of permutations of `xs ++ ys` in which *the first item of each permutation is from xs*; the remaining items of a permutation can be from the remaining items of `xs` and all of `ys`. Then all permutations of `xs` is `permute' xs [ ]`.

Function `permute'` applies recursion only on its first argument. Recall that `map (x:) xs` appends x as the head of each list in a list of lists.

```
permute' [ ] _      = [ [ ] ]
permute' [x] ys     = map (x:) (permute' ys [ ] )
permute' (x:xs) ys =   permute' [x] (xs ++ ys)
                    ++ permute' xs (x:ys)

permute xs = permute' xs [ ]
```

The definition is easy to understand for the first equation. The second equation says that if every permutation of elements of x and `ys` has to start with x , then any such permutation's head is x and tail is any permutation of `ys`. The third equation has two lists in its right side that are concatenated (they can be concatenated in either order). The first list is the set of permutations

that start with `x` and the second the permutations that start with the elements of `xs`.

```
*Main> permute "abc"
["abc", "bac", "bca", "acb", "cab", "cba"]
```

7.4.4 Refining a Recursive Solution: Towers of Hanoi

This is a classic problem that is commonly used in teaching recursion. I show the recursive solution, which is simple, but analyze the solution to arrive at more efficient solutions. Some of the solutions mimic an iterative solution, by storing the state information as argument values.

Given are three pegs, `S`, `D` and `I`, for *source*, *destination* and *intermediate*, respectively. There are n disks, $n > 0$, numbered 1 through n ; a larger-numbered disk has a larger size. Initially, all disks are stacked on peg `S` in order of increasing size from top to bottom, 1 at top and n at bottom. It is required to move all the disks to the destination peg, `D`, in a sequence of moves under the following constraints: (1) in each move the top disk of one peg is moved to the top of another peg, and (2) a larger disk is never put on top of a smaller disk on any peg.

Represent a move by a triple, (x, i, y) , where disk i is moved from the top of peg x to peg y . A solution is a list of moves. For $n = 1$, a single move, $[(S, 1, D)]$, constitutes the solution. The solution for $n = 2$, $[(S, 1, I), (S, 2, D), (I, 1, D)]$, is easy to see.

Simple recursive solution It is easy to solve the problem recursively. For a single disk, the solution is trivial. For multiple disks, argue as follows. There is a point at which the largest disk, n , is moved from `S` to another peg; for the moment assume that n is moved from `S` to `D`. At that time all other disks must be stacked on `I` because from constraint (1) disk n has to be the top disk on `S`, and from constraint (2) `D` cannot have any other disk on it so that n can be placed on its top. It follows that before the move of n all disks smaller than n have been moved from `S` to `I` and following the move of n all smaller disks have to be moved from `I` to `D`. Each of these subtasks, of moving the smallest $n - 1$ disks, can be solved recursively. In solving the subtasks, disk n may be disregarded because any disk can be placed on it; hence, its presence or absence is immaterial.

In Figure 7.9 `tower1 n x y z` specifies the number of disks n , and the source, destination and the intermediate pegs x , y and z , respectively.

A simple recurrence relation shows that the number of moves is $2^n - 1$. It can be shown that this is the smallest possible number of moves to solve this problem.

Improving the recursive solution Every function for this problem has to compute a list of $2^n - 1$ moves. Any improvement can result in only a linear speed-up.

Recursive solution of the tower of Hanoi

```

data Pegs = S | D | I
           deriving(Eq,Show)

tower1 n x y z
  | n == 0    = []
  | otherwise = (tower1 (n-1) x z y)
                ++ [(x,n,y)]
                ++ (tower1 (n-1) z y x)

*Main>    tower1 3 S D I
[(S,1,D),(S,2,I),(D,1,I),(S,3,D),(I,1,S),(I,2,D),(S,1,D)]

```

Figure 7.9**Improved solution of the tower of Hanoi**

```

tower2 n x y z
  | n == 0    = []
  | otherwise = first ++ [(x,n,y)] ++ (map replace first)
    where
      first = (tower2 (n-1) x z y)

shuffle a -- replace x by z, y by x, and z by y
  | a == x  = z
  | a == y  = x
  | a == z  = y

-- replace triple (x,m,y) by (z,m,x), for example
replace(u, m, v) = (shuffle u, m, shuffle v)

```

Figure 7.10

The recursive calls, `(tower1 (n-1) x z y)` and `(tower1 (n-1) z y x)`, generate similar sequences of moves. The latter sequence can be obtained from the former by replacing every occurrence of `x` by `z`, `y` by `x` and `z` by `y`. This strategy makes only one recursive call and generates the list corresponding to the second recursive call from the first one.

The solution in Figure 7.10 generates list `first` by the first recursive call. This list is modified by applying function `replace` to each of its elements so that `(x,m,y)` is replaced by `(z,m,x)`, and the other triples are similarly replaced.

Iterative solution There is an iterative solution for this problem in which in every step the top disks of two specific pegs are compared and the smaller disk moved to the other peg. This is a preferred solution for its efficiency, though $2^n - 1$ moves still have to be made. See Appendix 7.A.1 (page 453).

7.5 Recursive Data Types

I have so far emphasized recursion in the control structure of a program. Recursive data structures are equally important in order to specify data arrangement in a hierarchical manner. Naturally, recursive functions are the most appropriate way to process such data. In this section, I show several tree-like structures and functions defined over them.

The most important data structure that cannot be defined recursively is a graph. This is a serious obstacle in developing graph algorithms. There are techniques to impose tree structures on graphs to overcome this difficulty; see Breadth-first traversal in Section 6.6.1 (page 293) and Depth-first traversal in Section 6.6.2 (page 296).

7.5.1 Tree Structures

Many recursively defined data structures resemble a tree. I show definitions of a variety of trees in this section.

7.5.1.1 Binary Tree

Define a binary tree, `Bt`, whose nodes hold data of polymorphic type `a`. A binary tree either (1) is empty, denoted by `Null`, (2) has a single leaf node, denoted by `Leaf(a)`, or (3) is a tree that has a root node and two subtrees, denoted by `Node(Bt a, a, Bt a)`. The constructor functions `Leaf` and `Node` construct a tree from data items of the appropriate type. Note that we could have eliminated `Leaf` as a constructor, a tree that has a single node is a root node with two empty subtrees.

```
*Main> data Bt a = Null | Leaf(a) | Node(Bt a, a, Bt a)
*Main> :t Node
Node :: (Bt a, a, Bt a) -> Bt a
```

Figure 7.11 shows a description and a figure of a binary tree in which the data items at the nodes are integers.

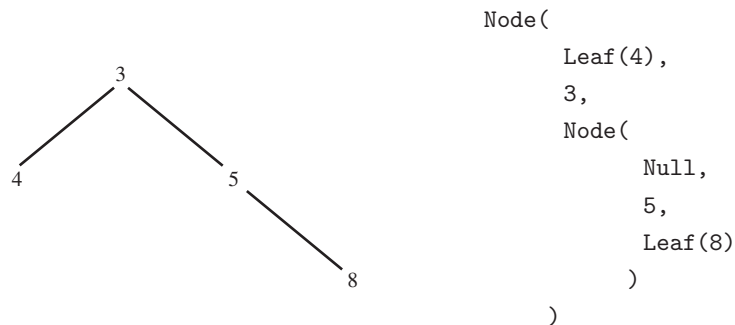


Figure 7.11 A binary tree with data values at nodes.

Functions can be defined as usual on recursive data structures. Below, function `find` returns `True` iff value `k` is in its argument binary tree.

```

find _ Null          = False
find k (Leaf(x))     = k == x
find k (Node(lt,v,rt)) = (find k lt) || (k == v) || (find k rt)
  
```

7.5.1.2 Binary Search Tree

A *binary search tree* (bst) can be used to implement a set on which there are two operations: (1) a membership test, that is, determine if a specific value is in the set, and (2) insert a value into the set. It is possible to delete values also, but I do not show that function here. The only constraint on values is that they should be totally ordered.

In a bst if a node has associated value `x`, all nodes in its left subtree have values smaller than `x` and in its right subtree values larger than `x`. Item `k` is added to tree `bst` using `insert k bst` and membership of `k` in `bst` is determined by using `find k bst`. An imperative implementation of this algorithm maintains a single mutable tree. The functional version in Figure 7.12 has no mutable variable, so each call to `insert` computes a new tree, although clever optimizations are often employed to eliminate most of the recomputing.

7.5.1.3 Full Binary Tree

Every non-leaf node in a *full* binary tree has exactly two children. Therefore, its definition can avoid the `Null` case for a general binary tree, `Bt`.

```

data Ftree a = Leaf(a) | Node(Ftree a, a, Ftree a)
  
```

Binary search tree

```

-- find a value

find _ Null      = False
find k (Leaf x) = k == x

find k (Node(left, x, right))
  | k == x = True
  | k < x  = find k left
  | k > x  = find k right

-- insert a value

insert k Null = Leaf(k)

insert k (Leaf x)
  | k == x = Leaf(x)
  | k < x  = Node(Leaf k, x, Null)
  | k > x  = Node(Null, x, Leaf k)

insert k (Node(left,x,right))
  | k == x = Node(left,x,right)
  | k < x  = Node(insert k left, x, right)
  | k > x  = Node(left, x, insert k right)

```

Figure 7.12

As a special case of *Ftree*, define *Etree* to represent an arithmetic expression over integer operands and the basic arithmetic operators, and define function *eval* that evaluates the expression represented by such a structure.

```

data Op = Sum | Prod | Diff | Quo | Rem
        deriving (Eq)
data Etree = Leaf(Int) | Nonleaf(Etree, Op, Etree)

eval (Leaf(x)) = x
eval (Nonleaf(e, op, f))
  | op == Sum = eval(e) + eval(f)

```

```

| op == Diff = eval(e) - eval(f)
| op == Prod = eval(e) * eval(f)
| op == Quo  = eval(e) `div` eval(f)
| op == Rem  = eval(e) `rem` eval(f)

```

As an example, the representation of expression $3 \times (2 + 4)$ as a full binary tree is

```
Nonleaf(Leaf(3), Prod, Nonleaf(Leaf(2), Sum, Leaf(4))) .
```

Applying `eval` to this tree returns the value 18.

7.5.1.4 General Tree

A tree in which non-leaf nodes may have an arbitrary number of children is a general tree (`Gtree`) (see Figure 7.13). Each node has an associated value of some base type. It may seem that the type declaration would be difficult, but it is surprisingly simple. A `Gtree` is a `Node` that is a tuple, a value of the base type (the value at the root of the tree) and a list of `Gtrees`, its subtrees. There is no explicit base case; a leaf node is simply a tuple with a value and an empty list of children.

```

data Gtree a = Node(a,[Gtree a])
instance Show a => Show (Gtree a) where
  show(Node(x, [])) = show x
  show(Node(x, cs)) = show x ++ show cs

```

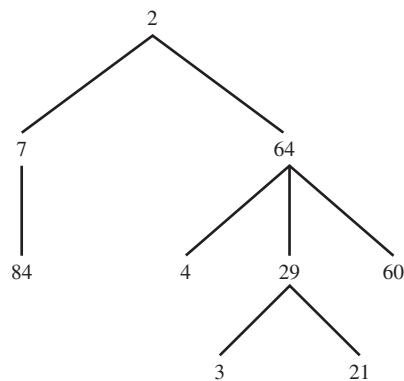


Figure 7.13 A general tree (`Gtree`).

Function `show` applied to a tree displays it according to its structural description, value of a node followed by a list of its subtrees. The tree in Figure 7.13, whose declaration is in Figure 7.14, is shown as `2[7[84],64[4,29[3,21],60]]`. The sequence of values, ignoring the square brackets, are in *preorder*; see Section 6.6.2.1 (page 296).


```

Node(2,
  [Node(7, [
    Node(84, []) -- leaf node with 84 has an empty children list
  ]), -- end of subtree rooted at 7
  Node(64, [ -- start of subtree rooted at 64
    Node(4, []), -- leaf node with 4
    Node(29, [ -- start of subtree rooted at 29
      Node(3, []), -- leaf node with 3
      Node(21, []) -- leaf node with 21
    ]), -- end of subtree rooted at 29
    Node(60, []) -- leaf node with 60
  ]) -- end of subtree rooted at 64
]) -- end of the whole tree rooted at 2

```

Figure 7.14 Declaration of the Gtree in Figure 7.13.

7.5.2 Tree Isomorphism

Define two Gtrees to be *isomorphic* if they are identical up to a permutation of the children of each node. See Figure 7.15 for isomorphic trees. I develop algorithms for this problem and some of its generalizations.

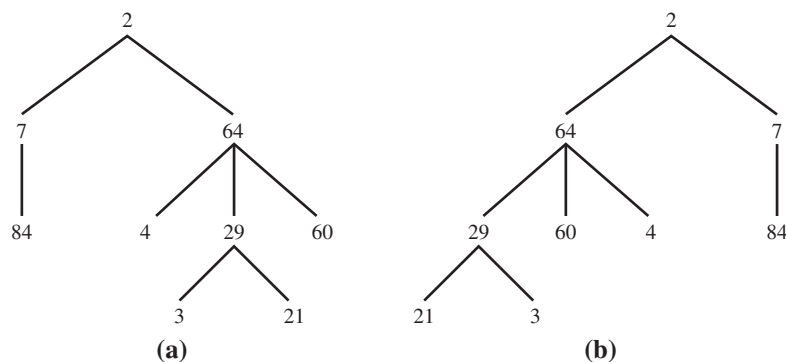


Figure 7.15 Isomorphic trees.

A simple application of isomorphism is in comparing the directories of a hierarchical file system. A *directory*, or *folder*, is either a *file* or a directory containing some number of directories. There is a single *root* directory. So, the file system is a tree in which the files are leaf nodes, folders are internal nodes, and every node has the name of the associated directory. Directory names may not be distinct because sub-directories of different directories may have the same name. There is no order of files within a directory. Two such trees are isomorphic if their directory structures are identical.

This problem can be extended to compare arithmetic and logical expressions for isomorphism. Then the nodes are operators and the leaves are operands. Only the children of a commutative operator can be permuted. I solve a few variations of this problem.

7.5.2.1 Simple Isomorphism

First, I consider the problem of comparing directory structures. In abstract terms, given are two Gtrees, x and y , as defined in Section 7.5.1.4. Thus, a node has the form (v, vs) , where v is a value of some given base type and vs a list of trees. Trees (v, vs) and (v', vs') are isomorphic if (1) $v == v'$, and (2) there is a permutation of the elements of vs' , say vs'' , such that corresponding elements of vs and vs'' are isomorphic. Note that there is no special requirement for the base case because the list associated with a leaf node is empty and empty lists are isomorphic.

At first glance the problem seems formidable. But there is a rather simple solution that requires only an understanding of recursion. Let us start with a simpler problem. Suppose we want to determine if two lists of some base type are permutations of each other. The naive algorithm searches for each element of one list in the other. A better algorithm is to sort each list, provided there is a total order over the items, and check if the sorted lists are equal. This algorithm exploits a fundamental fact of isomorphism, that it is an equivalence relation. Each equivalence class has a unique representative, namely the sorted list. Call the result of sorting a list its *normal form*.

We adapt this algorithm for the tree isomorphism problem. Convert each tree to its normal form and then compare the two normal forms for equality to ascertain if the trees are isomorphic. To convert a tree to a normal form, first convert each of its subtrees to normal form and then sort the list of subtrees.

I show the algorithm below. Assume that a total order exists over the base type a . Augment the definition of Gtree to define equality and total order over trees. Function `sort` is any polymorphic sorting function that can be applied to a list of normalized trees. In particular, `mergesort` of Section 7.2.8.3 (page 400) can be used in place of `sort`. Function `norm` computes the normal form of its argument. Function `isomorph` tests two trees for isomorphism.

```
data Gtree a = Node(a, [Gtree a])
              deriving (Eq, Ord)

norm (Node(v, children)) = Node(v, sort(map norm children))

isomorph g h = (norm g) == (norm h)
```

Observe the importance of polymorphic data types and type classes in this solution. We expect `sort` to sort any set of data items provided a total order is given, as it is for `Gtrees`.

7.5.2.2 Isomorphism of Expressions

I extend the isomorphism algorithm to check for isomorphism of expressions. An expression is represented by a tree in which the internal nodes have operators and leaves have operands. Expressions over any domain can be compared. I show the example for propositional logical expressions with “and” ($\wedge$, `And`), “or” ($\vee$, `Or`), “not” ($\neg$, `Not`), “implication” ($\Rightarrow$, `Impl`) and “exclusive or” ($\oplus$, `Eor`) as the operators, and strings, representing variables, as operands. Below, `Ops` defines the operators and `Etree` the expressions.

Call an operator a *com*-operator if the operator can be applied to its operands in arbitrary order. A commutative operator over two operands is a *com*-operator, but over multiple operands a *com*-operator has to be both commutative and associative.

Expression isomorphism with commutativity

```
data Ops = And | Or | Not | Impl | Eor
         deriving (Eq,Ord)

data Etree = Leaf String | Nonleaf(Ops, [Etree])
           deriving (Eq,Ord)

com And  = True
com Or   = True
com Eor  = True
com Not  = False
com Impl = False

norm (Leaf x) = Leaf x
norm (Nonleaf(v, subexps))
  | com(v)     = Nonleaf(v, sort(map norm subexps))
  | otherwise  = Nonleaf(v, (map norm subexps))

isomorph g h = (norm g) == (norm h)
```

Figure 7.16

Define two expressions to be isomorphic if they are identical up to permutations of operands of their *com*-operators. Thus, $p \wedge q \wedge r$ and $q \wedge r \wedge p$ are isomorphic whereas $p \Rightarrow q$ and $q \Rightarrow p$ are not. In Figure 7.16 function `com` defines the *com*-operators. The normal form of an expression is obtained by normalizing each operand of an operator and sorting the list of operands *only if* the operator is a *com*-operator. Function `norm` has been redefined in Figure 7.16.

Isomorphism merely compares two expressions for structural properties related to permutation. It does not apply distributivity or De Morgan's law in comparing expressions. For a full equality check, each expression should be converted to conjunctive or disjunctive normal form and then checked for isomorphism.

7.5.2.3 Isomorphism of Expressions Under Idempotence

A small, but useful, extension of the last algorithm is to consider idempotence of the operators. An operator, such as “and”, is idempotent because a pair of

Expression isomorphism with commutativity and idempotence

```

uniq [ ] = [ ]
uniq [x] = [x]
uniq (x:y:xs) = if x == y then uniq (y:xs) else x: uniq(y:xs)

idem And  = True
idem Or   = True
idem Eor  = False
idem Not  = False
idem Impl = False

norm (Leaf x) = Leaf x

norm (Nonleaf(v, gts))
  | com(v) && idem(v)           = Nonleaf(v, uniq(sort(map norm gts)))
  | com(v) && (not (idem(v)))    = Nonleaf(v, sort(map norm gts))
  | otherwise                   = Nonleaf(v, (map norm gts))

isomorph g h = (norm g) == (norm h)

```

Figure 7.17

isomorphic operands can be replaced by just one occurrence of the operand. Typically, an idempotent operator is commutative but not conversely. For example, “exclusive or” is commutative but not idempotent.

The program in Figure 7.17 replaces the operand list of every idempotent operator by a list of its unique elements in sorted order. Function `uniq` removes repeated elements from a list assuming that its argument list is sorted. Function `idem`, similar to `com`, identifies the idempotent operators. The program assumes that there is no non-commutative idempotent operator, which is true for our set of operators. Function `norm` is redefined in Figure 7.17. I use the definition of `com` from Section 7.5.2.2.

7.6 Dynamic Programming

7.6.1 Top-Down versus Bottom-Up Solution Procedures

7.6.1.1 Top-Down Solution Procedures

At the beginning of this chapter, I outlined a top-down approach to problem solving: the original problem is divided into component subproblems that are further subdivided similarly. The smallest subproblems (there could be several smallest ones) have known solutions. Then the solutions of component subproblems are combined to yield solutions to the original problem. A solution procedure specifies both the method of division of problems into subproblems and the method of combination of subproblem solutions to yield the solution of the original problem. The programs written in Haskell exemplify this approach. Several problems in data parallel programming, such as the Fast Fourier Transform and Batchers sort, that are treated in Chapter 8 are also described in the top-down recursive style.

7.6.1.2 Bottom-Up Solution Procedures

There is a bottom-up procedure that mimics the top-down in reverse. First, identify *all* the subproblems of the original problem that will be solved during the top-down procedure. Define a “smaller” relation among subproblems as follows: call problem Q smaller than P if Q is called during a recursive top-down solution of P . It is easy to see which subproblems are directly called from P , so first identify them as being smaller. Since “smaller” is transitive, all smaller subproblems of Q are smaller than P . Relation “smaller” is a partial and well-founded order over the subproblems if the recursive calls terminate.

Next, postulate a topological order $<$ over the subproblems so that if Q is smaller than P then $Q < P$. Call Q “lower” than P and P “higher” than Q if $Q < P$. We have seen in Section 3.4.7 (page 111) that a topological order corresponding to a partial

order over a finite set always exists. A smaller subproblem is lower, but a lower subproblem may not, necessarily, be smaller.

The bottom-up procedure solves the subproblems in sequence according to the total order $<$, from the lowest to the highest. Store the solution for each subproblem as it is solved. The major observation on which the bottom-up procedure relies is that whenever subproblem P needs to be solved, it is either (1) the lowest one according to $<$, for which the solution is known, or (2) one whose subproblems, which are lower in the ordering, have already been solved; so their solutions are available to be combined to yield the solution to P . The bottom-up procedure is usually iterative, not recursive.

7.6.1.3 When is Bottom-Up Preferable to Top-Down?

A top-down procedure may require solving the same subproblem multiple times; these are called “overlapped” subproblems. The bottom-up procedure avoids solving overlapped subproblems more than once. And the overhead of stack storage in a top-down scheme is replaced by storage of solutions of certain subproblems during the computation, which may be substantially less. We shall see examples in which top-down procedures require exponential time and space whereas bottom-up procedures require polynomial amounts of each.

In some cases, top-down may be the preferred scheme because a tail-recursive definition is, typically, implemented by iteration without incurring the overhead of a stack and the top-down description of an algorithm may be more transparent.

To motivate the need for bottom-up solutions, consider the computation of the n^{th} number in the Fibonacci sequence using the definition given below.

```
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

Computation of `fib 10` calls `fib 9` and `fib 8`, then computation `fib 9` calls `fib 8` again, so `fib 8` is computed twice. As shown in Exercise 10 (page 447) during the computation of `fib n`, $n > 1$, `fib m` for any m , $0 < m < n$, is called F_{n+1-m} times. We avoided this problem in Section 7.2.5.3 (page 388) by computing a pair of consecutive Fibonacci numbers in each step. The code looks nominally top-down though the tail-recursion will typically be implemented by iteration.

Partitions of a positive integer I consider a more involved problem and compare its top-down and bottom-up solutions. Enumerate all partitions of a given positive integer, where partition of n is a set of *distinct* positive integers whose sum is n .

Represent each partition by a list of integers in increasing order. Define a more general function, `parts n i`, where i is a positive integer, that partitions n so that

each number in any partition is at least i . Then `partition n` is just `parts n 1`. Define `parts n i` using case analysis: either the first number in a partition is i or greater than i . If i is in a partition, the remaining numbers in the partition are all greater than i and their sum is $n-i$. If i is not in a partition, then all numbers in the partition are greater than i and their sum is n .

```
parts n i
  | n < i      = []
  | n == i     = [[n]]
  | otherwise =
      (map (i:) (parts (n-i) (i+1)))
      ++ (parts n (i+1))

partition n   = parts n 1
```

The value of `partition 8` is `[[1,2,5],[1,3,4],[1,7],[2,6],[3,5],[8]]`.

The top-down recursive procedure seems to be efficient in that it divides the problem neatly so that the subproblems do not seem to overlap. Actually, it is very inefficient; there is a very large number of calls with the same values of the arguments, n and i . For `partition 60`, there are nearly 90,000 repeated calls where $n > i$ so the `otherwise` clause is invoked. The number of times `map (i:)` is applied is 182,205.

In contrast, a bottom-up solution for `partition N` starts by identifying the subproblems, all the pairs (n, i) where $1 \leq n \leq N$ and $1 \leq i \leq n$. There are $\mathcal{O}(N^2)$ such pairs, exactly 1,860 for $N = 60$. We can order the pairs in a variety of ways, but from the given recursive solution a pair with a larger value of the second component is taken to be smaller. That is, $(n, i) < (m, j)$ if $i > j$. This is a partial order over pairs of positive integers, not a total order, because it imposes no order on pairs with the same value of the second component, such as $(30, 12)$ and $(40, 12)$. Since the second component is at most N , this is a well-founded order. I show a solution in Figure 7.18.

The partitions for each pair (n, i) are stored in matrix *partitions*. Each partition is a non-empty set, and all partitions are a set of sets, similar to the list of lists in the recursive solution. The initialization part of the program is not shown explicitly; it stores `{}` in `partitions[n, i]` if $n < i$, and `{n}` if $n = i$, as shown in its postcondition.

The solution in Figure 7.18 is entirely iterative, determined by the well-founded order described above, $(n, i) < (m, j)$ if $i > j$. Therefore, the outer `for` loop computes over i from $N - 1$ down to 1. The inner `for` loop computes in arbitrary order over the set $\{j \mid 1 \leq j \leq N\}$, though any execution on a sequential computer has to impose a total order.

Partitions of a positive integer

```

Proc parts(n,i)
  for i ∈ rev(1..N) do { rev(1..N) = ⟨N, N − 1, ⋯, 1⟩ }
    for n ∈ {j | 1 ≤ j ≤ N} do
      partitions[n,i] := ({i} ∪ {S | S ∈ partitions[n − i, i + 1]})
                        ∪ partitions[n, i + 1]
    endfor
  endfor
endProc

initialize;
{ (∀n, i : 1 ≤ n < i ≤ N : partitions[n, i] = {}),
  (∀n : 1 ≤ n ≤ N : partitions[n, n] = {{n}} }
parts(N, 1)

```

Figure 7.18

There are many possible optimization strategies for storage. For example, in the outer **for** loop only the columns *i* and *i* + 1 of *partitions* are needed during an iteration. So, columns of higher index can be output and discarded.

7.6.1.4 Encoding Bottom-Up Solutions in Haskell; Memoization

The key feature of a bottom-up solution is that solving a subproblem requires quick access to solutions of smaller subproblems. The data structure employed for solution is, typically, an array-like structure, possibly of a higher dimension, that permits efficient random access. A functional programming language like Haskell does not permit efficient random access. So, most dynamic programming solutions are encoded in imperative style.

Functional programming languages sometimes allow defining *memo functions* that save function values of calls so that they are not recomputed. Memoization is simply a caching technique, implemented under the direction of the programmer. It is possible to define memo functions in Haskell, but I do not cover that topic here. In most cases, explicit control over the cached values is required to achieve both storage and computing time efficiency.

7.6.2 Optimality Principle

Optimization problems Optimization problems typically have a number of *feasible* solutions, that is, the ones that satisfy certain constraints specific to the problem. An *objective function* is defined over solutions whose value is a number.

The goal is to find an *optimal solution*, a feasible solution that has the largest (smallest) value of the objective function for a maximization (minimization) problem. For example, problems related to finding paths between two points in a road network may have the objective of finding the best path according to some criterion, such as the distance, the time required to travel the path or the amount of toll to be paid. A feasible solution is a sequence of roads connecting the source and the destination in the given road network. Each of the given measures is an objective function, and the desired solution is a feasible solution that minimizes the value of the objective function.

Optimality principle Dynamic programming for optimization problems depend on the *Optimality principle*: the optimal solution of a problem can be constructed by combining the optimal solutions of some of its subproblems. For example, a shortest path from point A to point B either follows a single road without any intermediate point or it goes through some point C. In the latter case, the paths from A to C and from C to B themselves have to be the shortest paths. Otherwise, we can replace a non-shortest path by a shortest path, thus getting an even shorter solution, a contradiction. The identity of C would not be known *a priori*. So all points in the road network are candidates to become C. However, the search space is now considerably reduced because we do not have to enumerate *all* paths, just the ones that are optimal for certain designated pairs of points.

The optimality principle is valid for many optimization problems, but not all; I show one such example later in this section.

Top-down formulation of dynamic programming solution Dynamic programming is a bottom-up solution procedure for problems in which the optimality principle is applicable. It was observed by Bellman [1957], the inventor of dynamic programming, that many optimization problems can be formulated as a top-down recursive procedure, using the optimality principle, and solved efficiently in a bottom-up fashion. Dynamic programming has been applied on a vast number of problems, in diverse areas such as biology, control theory, economics and computer science. We study a few small examples of relevance in computer science.

A dynamic programming formulation starts with a top-down strategy that is later implemented in bottom-up fashion for efficiency. The top-down strategy yields a set of subproblems. The bottom-up strategy finds a way to enumerate the subproblems in an iterative manner so that a subproblem is enumerated only after all its subproblems have been enumerated. This may be the most inventive step in the solution procedure. In the road network problem this amounts to identifying all possible points C through which a shortest path from A to B may pass. For each decomposition, recursively compute the optimal solutions of the subproblems and

combine those solutions to arrive at a *conditional* optimal solution of the original problem, the best possible solution under the given decomposition. Now, choose the best conditional optimal solution, over all decompositions, to arrive at the true optimal solution.

In many cases in this chapter, I do not compute the optimal solution, just the value of the objective function for the optimal solution. This computation, typically, is simpler, and allows development of the initial version of the program. It is quite easy to modify this program to compute the optimal solution itself.

Note that the bottom-up procedure may solve certain subproblems that may never be solved in the top-down procedure. If the number of such subproblems is small, the gain in efficiency in bottom-up will outweigh the wasted effort.

The termination problem There is an important requirement for the top-down procedure, it must terminate, that is, the decomposition should always result in strictly smaller subproblems. As we have seen in reasoning about recursive programs (Section 7.3, page 406), correctness of a program is contingent on its termination. The road network problem is an excellent example of non-termination. In decomposing a shortest path problem from A to B that goes through C, the path problems (A to C) and (C to B) have to be smaller in size by some well-founded measure, say the length of the shortest path. In this case they are not smaller because problem (A to C) may pick B as an intermediate point, so it needs the solution of problem (A to B), a circularity. So, the shortest path problem is not solvable by the given decomposition, though it obeys the optimality principle.

Where dynamic programming is inapplicable Not all combinatorial optimization problems can be solved by dynamic programming because the optimality principle may not be applicable. A simple example is computing the optimal *addition chain*. An addition chain for a positive integer n is an increasing sequence of integers starting with 1 in which every element is a sum of two previous elements of the sequence (which need not be distinct), and the sequence ends with n . An addition chain for $n = 45$ is shown below in which the summands for each element are shown next to the element:

$\langle 1, 2 : 1+1, 4 : 2+2, 5 : 4+1, 10 : 5+5, 11 : 10+1, 22 : 11+11, 44 : 22+22, 45 : 44+1 \rangle$.

This addition chain has length 8, counting the number of additions performed. An optimal addition chain for a given value has the minimum possible length. An optimal addition chain for 45, with 7 additions, is:

$\langle 1, 2 : 1 + 1, 4 : 2 + 2, 5 : 4 + 1, 10 : 5 + 5, 20 : 10 + 10, 40 : 20 + 20, 45 : 40 + 5 \rangle$.

It seems that finding an optimal chain would be amenable to problem decomposition. If n is computed by adding s and t in the optimal addition chain, then we may believe that the chain would include the optimal addition chains for s and t . That is not necessarily the case because the optimal addition chains for s and t need not be disjoint. In particular, s and t may be equal, as is the case for several entries in both chains. Therefore, the optimality principle does not apply. In contrast, the problem we study next, matrix chain multiplication, has a similar decomposition and is solvable by dynamic programming because the optimality principle applies.

7.6.3 Matrix Chain Multiplication

Consider the problem of multiplying a sequence of matrices. The straightforward algorithm for multiplication of matrices of dimensions $p \times q$ and $q \times r$ takes $p \times q \times r$ steps and yields a matrix of dimension $p \times r$. Since matrix multiplication is associative, the matrix product $P \times Q \times R$ can be computed either as $P \times (Q \times R)$ or $(P \times Q) \times R$. However, writing the dimensions of a matrix as its subscripts, $P_{a \times b} \times (Q_{b \times c} \times R_{c \times d})$ takes $bcd + abd$ steps whereas $(P_{a \times b} \times Q_{b \times c}) \times R_{c \times d}$ takes $abc + acd$ steps, and the number of steps could be vastly different. For example, with $a, b, c, d = 10^1, 10^2, 10^1, 10^3$, the first scheme takes $10^6 + 10^6$ steps and the second scheme only $10^4 + 10^5$ steps. The differences could be even more pronounced when a long chain of matrices of substantially different dimensions are multiplied.

Let M_i , $0 \leq i < n$, be a matrix having r_i rows and r_{i+1} columns. The problem is to find the best scheme, that is, the one using the fewest steps, for computing $M_0 \times \cdots \times M_i \times \cdots \times M_{n-1}$. Use P_{ij} to denote the product of the matrices over interval (i, j) , that is, from M_i to M_{j-1} , for $0 \leq i < j < n$, and c_{ij} for the cost, that is, the minimum number of steps, needed to compute P_{ij} . The goal is to devise a procedure to compute c_{0n} . I assume nothing specific about the values of the matrices M_i . For example, if all M_i are equal, that is, the goal is to compute M^n for some matrix M , there is a simple algorithm—see Section 8.6.2.5 (page 497).

The number of possible multiplication schemes is given by Catalan numbers which grow exponentially; the n^{th} Catalan number is $\frac{(2n)!}{n!(n+1)!}$. So, choosing the best scheme by exhaustive enumeration of all possible schemes is possible only for small values of n . But there is an excellent dynamic programming algorithm for this problem. Suppose the best scheme computes P_{0n} as $P_{0k} \times P_{kn}$, that is, the last multiplication is between these two matrices. Then both P_{0k} and P_{kn} must, in their turn, be computed by the best schemes, otherwise, the overall scheme would not be optimal. This is the optimality principle employed by the algorithm.

The cost of computing P_{0n} as $P_{0k} \times P_{kn}$ is $c_{0k} + c_{kn} + r_0 \cdot r_k \cdot r_n$, the first two terms being the costs of computing P_{0k} and P_{kn} optimally and the last term for

multiplying them (the dimension of P_{0k} is $r_0 \times r_k$ and of P_{kn} is $r_k \times r_n$). The identity of k is unknown; so, the minimum value of the cost expression has to be computed over all k to determine c_{0n} :

$$c_{i(i+1)} = 0, \text{ and} \\ c_{ij} = (\min k : i < k < j : c_{ik} + c_{kj} + r_i \cdot r_k \cdot r_j), \text{ for } i + 1 < j.$$

The value of k at which the minimum is attained is the *split point* of interval $(i..j)$. This formulation immediately gives a top-down recursive algorithm for computing c_{0n} . However, the top-down procedure will solve the problem over many intervals repeatedly.

For the bottom-up procedure, observe that for each interval $(i..j)$ recursive calls are made to its subintervals, $(i..k)$ and $(k..j)$, for each k , $i < k < j$. Therefore, the subproblems to be solved are described by the set of intervals. Let $cost[i..j]$ store c_{ij} . Solve the intervals in order of their lengths. So, the computation of $c_{ik} + c_{kj} + r_i \cdot r_k \cdot r_j$, which amounts to computing $cost[i..k] + cost[k..j] + r_i \cdot r_k \cdot r_j$, just takes two additions and two multiplications of integers. The number of intervals is $\mathcal{O}(n^2)$ and the entire computation takes $\mathcal{O}(n^2)$ time and space.

To compute the optimal multiplication scheme, store the split point for interval $(i..j)$ in $split[i..j]$. This is the value of k where $c_{ik} + c_{kj} + r_i \cdot r_k \cdot r_j$ is minimum. Then the computation scheme for P_{0n} is $P_{0m} \times P_{mn}$ where $split[0..n] = m$. And recursively find all the split points in computing P_{0m} and P_{mn} in order to obtain the optimal scheme.

7.6.4 Longest Common Subsequence

The longest common subsequence (lcs) of a non-empty set of sequences is the longest sequence that is a subsequence of all sequences in the set. As an example, the sequences "12345678" and "452913587" have common subsequences, "458", "2358", "1357" and "1358". Each of the last three is an lcs.

I consider the problem of computing the lcs of two sequences. The solution can be extended to a set by replacing any two sequences in the set by their lcs and then repeating the algorithm. This is a problem of interest because it arises in several applications in string processing. For example, the `diff` command of UNIX that compares two documents for their differences uses the algorithm developed here.

The top-down solution is surprisingly simple. Below, use the list notation for sequences. Write $lcs(us, vs)$ for the lcs of lists us and vs . The type of items in the lists are irrelevant as long as they are the same. Clearly, $lcs(us, []) = [] = lcs([], vs)$. If neither sequence is empty, say of the form $(u:us)$ and $(v:vs)$, consider the following two possibilities.

- $u = v$: Let $ws = lcs(u:us, u:vs)$. I show that ws includes u as its first element. The proof is by contradiction. If ws does not include u , it is a common subsequence of us and vs . Now $(u:ws)$ is a longer common subsequence of $(u:us)$ and $(u:vs)$, contradicting that ws is the lcs. So, $lcs(u:us, u:vs) = u:lcs(us, vs)$.
- $u \neq v$: Not both of u and v are included in $lcs(u:us, v:vs)$. If u is not included, then $lcs(u:us, v:vs) = lcs(us, v:vs)$. Similarly, if v is not included, $lcs(u:us, v:vs) = lcs(u:us, vs)$. The lcs of $lcs(u:us, v:vs)$ is the longer of these two.

I encode this scheme in Haskell in Figure 7.19:

Longest common subsequence in Haskell

```

lcs us [ ] = [ ]
lcs [ ] vs = [ ]

lcs (u:us) (v:vs)
  | u == v = u: (lcs us vs)
  | otherwise = if (length first) > (length second)
                 then first else second
                 where
                   first  = lcs (u:us) vs
                   second = lcs us (v:vs)

```

Figure 7.19

7.6.4.1 Dynamic Programming Solution

The top-down recursive solution requires computation of $lcs(us, v:vs)$ and $lcs(u:us, vs)$ in order to compute $lcs(u:us, v:vs)$ when $u \neq v$, and they both require computation of $lcs(us, vs)$ in the next recursive call. It can be shown quite easily that the given procedure requires solving an exponential number of subproblems.

I derive a bottom-up solution that takes polynomial time. For the explanation below, I use strings of characters for the lists though any other type will work as well. Observe in the top-down solution that recursive calls are made to the suffixes of both lists. Therefore, compute the function values for all pairs of suffixes in the order such that any computation step requires looking up some precomputed values and doing a fixed number of arithmetic and comparisons operations. For strings of lengths m and n , the number of pairs of suffixes is $\mathcal{O}(m \cdot n)$, so the computation time is also bounded by this function.

Represent us and vs by arrays of characters, of lengths m and n , respectively. Number the positions in each array starting at 0, so $us[i]$ is the character at position i . Write us_i for the suffix of us starting at position i , $0 \leq i \leq m$. The length of us_i is $m - i$, so us_m is the empty string and $us_0 = us$. Define vs_j , $0 \leq j \leq n$, similarly. I compute matrix $lcsmat$ where $lcsmat[i, j] = lcs(us_i, vs_j)$, $0 \leq i \leq m$ and $0 \leq j \leq n$.

The computation in Figure 7.20 proceeds from the smallest subproblem to the largest, as is the norm in dynamic programming. The top-down algorithm imposes the following order on pairs which the given program obeys: the subproblems that are directly smaller than (i, j) are $(i + 1, j)$, $(i, j + 1)$ and $(i + 1, j + 1)$; so, define any pair (p, q) to be smaller than (p', q') if $p + q > p' + q'$. The smallest subproblems are the ones in which one of the sequences is empty. That is, $lcsmat[m, j]$ for all j , $0 \leq j \leq n$, and $lcsmat[i, n]$ for all i , $0 \leq i \leq m$, each corresponding to an empty sequence.

Longest common subsequence

```

{ Initialize }
{ Corresponding to lcs us [ ] = [ ] }
  for  $i \in rev(0..m+1)$  do  $lcsmat[i, n] := []$  endfor ; {  $rev(0..m+1) = \langle m+1, m, \dots, 0 \rangle$  }
{ Corresponding to lcs [ ] vs = [ ] }
  for  $j \in rev(0..n+1)$  do  $lcsmat[m, j] := []$  endfor ; {  $rev(0..n+1) = \langle n+1, n, \dots, 0 \rangle$  }

{ Computation proper }
for  $i \in rev(0..m)$  do
  for  $j \in rev(0..n)$  do
    { Below, : is concatenation, Cons of Haskell. }
    if  $us[i] = vs[j]$  then  $lcsmat[i, j] := us[i]:lcsmat[i+1, j+1]$ 
    else {  $us[i] \neq vs[j]$  }
      first :=  $lcsmat[i, j+1]$ ;
      second :=  $lcsmat[i+1, j]$ ;
      if  $length(first) > length(second)$ 
        then  $lcsmat[i, j] := first$ 
        else  $lcsmat[i, j] := second$ 
      endif
    endif
  endfor
endfor
endfor

```

Figure 7.20

The computation time is $\mathcal{O}(m \times n)$. And space requirement is for the storage of $m \times n$ subsequences each of which can be at most $\max(m, n)$ in length. Since our only interest is in the matrix entry $lcsmat[0, 0]$, we can reduce the space requirement considerably in two different ways: (1) store only the part of the matrix that is needed for subsequent computation, and (2) store the information needed to recreate the subsequence at any $lcsmat[i, j]$ instead of the entire subsequence itself. These schemes are further explored in Appendix 7.A.2 (page 456), which show that $\mathcal{O}(m \times n)$ space suffices.

7.6.5 Minimum Edit Distance

This problem is a significant generalization of the longest common subsequence problem of Section 7.6.4 (page 439). The problem is to transform a string us to string vs in a sequence of steps by applying any of the following operations in each step: (1) insert a symbol anywhere in us , (2) delete any symbol of us , or (3) replace a symbol of us by another symbol.

For example, we may transform $pqspt$ to qps in the following way, writing $i(x)$ to denote insertion of symbol x , $d(x)$ for deletion of one instance of x , and $r(x, y)$ for replacement of one instance of x by y , where the relevant changes are underlined: $pqspt \xrightarrow{d(p)} qspt \xrightarrow{r(s,p)} qppt \xrightarrow{i(s)} qpspt \xrightarrow{d(p)} qpst \xrightarrow{d(t)} qps$. Another transformation, taking fewer steps, is: $pqspt \xrightarrow{d(p)} qspt \xrightarrow{d(s)} qpt \xrightarrow{r(t,s)} qps$.

Our goal is to find the minimum number of steps for such a transformation. More generally, assign a positive cost to each step, c_i , c_d and c_r , for insert, delete and replace. The goal is to find the sequence of steps that has the minimum total cost. This cost is known as “Levenshtein distance” or “edit distance” between a pair of strings.

Let $ed(us, vs)$ be the edit distance between us and vs . It can be shown that edit distance satisfies the properties of a metric that is familiar from geometry: (1) $ed(us, vs) \geq 0$, (2) $ed(us, vs) = 0$ iff $us = vs$, and (3) (triangle inequality) $ed(us, vs) + ed(vs, ws) \geq ed(us, ws)$.

If $c_i = c_d$, every operation has an inverse operation of equal cost. Then $ed(us, vs) = ed(vs, us)$, by replacing $i(x)$ and $r(x, y)$ in one by $d(x)$ and $r(y, x)$, respectively, in the other.

There are other kinds of edit distances that include more transformation operations. Such string transformation problems have important applications in computational biology and string matching.

7.6.5.1 Underlying Mathematics

In the following solution, c_i and c_d need not be equal. Let $us \xrightarrow{p} vs$ denote that p is an optimal sequence of steps to transform us to vs , so the cost of p is $ed(us, vs)$.

Consider three cases:

- One of the strings, us or vs , is empty: If vs is empty, then delete all symbols of us one by one, so $ed(us, []) = c_d \times |us|$ where $|us|$ is the length of us . Similarly, if us is empty, insert each symbol of vs in us , so $ed([], vs) = c_i \times |vs|$. So, if both strings are empty, the cost is 0.

- Both strings are non-empty, and they have equal first symbols: Then the strings are of the form $u:us$ and $u:vs$. Transform us to vs with the minimum possible cost using $us \xrightarrow{p} vs$; then $u:us \xrightarrow{p} u:vs$ is a minimum cost transformation of the strings. Therefore, $ed(u:us, u:vs) = ed(us, vs)$.

- Both strings are non-empty, and they have unequal first symbols: Consider strings of the form $u:us$ and $v:vs$ where $u \neq v$. There are three possible ways to transform $u:us$ to $v:vs$.

(1) Delete symbol u and transform us to $v:vs$, that is, $u:us \xrightarrow{d(u)} us \xrightarrow{p} v:vs$. The cost of this transformation is $c_d + ed(us, v:vs)$.

(2) Insert symbol v and transform $u:us$ to vs , that is, $u:us \xrightarrow{i(v)} v:u:us \xrightarrow{p} v:vs$. The cost of this transformation is $c_i + ed(u:us, vs)$.

(3) Replace u by v and then transform us to vs , that is, $u:us \xrightarrow{r(u,v)} v:us \xrightarrow{p} v:vs$. The cost of this transformation is $c_r + ed(us, vs)$.

Therefore, the edit distance is given by the minimum cost over (1,2,3) when $u \neq v$, that is,

$$ed(u:us, v:vs) = \min(c_d + ed(us, v:vs), c_i + ed(u:us, vs), c_r + ed(us, vs)).$$

7.6.5.2 A Program to Compute Edit Distance

The Haskell program in Figure 7.21 uses `cd`, `ci` and `cr` for c_d , c_i and c_r . Value of function `minimum` over a list is the minimum value in the list.

7.6.5.3 A Dynamic Programming Solution

The top-down solution, above, requires computation of $ed(us, vs)$, $ed(us, v:vs)$ and $ed(u:us, vs)$ in order to compute $ed(u:us, v:vs)$. Its structure is very similar to that for the longest common subsequence problem, and it requires exponential time. As in that problem, observe that calls are always made to suffixes of both lists. Therefore, compute the function values for all suffixes in the same order as for the lcs problem.

The representations of us and vs are exactly as they are for the longest common subsequence problem. I repeat the description. Represent the given strings us and

Minimum edit distances in Haskell

```

ed us [ ] = cd * (length us)
ed [ ] vs = ci * (length vs)

ed (u:us) (v:vs)
| u == v = ed us vs
| otherwise = minimum
    [cd + (ed us (v:vs)) , -- delete
     ci + (ed (u:us) vs), -- insert
     cr + (ed us vs)      -- replace
    ]

```

Figure 7.21

Edit distance computation

```

{ Initialize }
{ Corresponding to ed us [ ] = cd * (length us) }
  for  $i \in rev(0..m)$  do  $cost[i, n] := cd * (m - i)$  endfor ; {  $rev(0..m) = \langle m, m-1, \dots, 0 \rangle$  }
{ Corresponding to ed [ ] vs = ci * (length vs) }
  for  $j \in rev(0..n)$  do  $cost[m, j] := ci * (n - j)$  endfor ; {  $rev(0..n) = \langle n, n-1, \dots, 0 \rangle$  }

{ Computation proper }
  for  $i \in rev(0..m)$  do
    for  $j \in rev(0..n)$  do
      {  $I :: (\forall (i', j') : (i', j') < (i, j) : cost[i', j'] = ed(us_{i'}, vs_{j'}))$  }
      if  $us[i] = vs[j]$  then  $cost[i, j] := cost[i+1, j+1]$ 
      else  $cost[i, j] :=$ 
         $\min(cd + cost[i+1, j], ci + cost[i, j+1], cr + cost[i+1, j+1])$ 
      endif
    endfor
  endfor
endfor

```

Figure 7.22

us as arrays of characters of length m and n , respectively. Number the positions in each array starting at 0 and write us_i for the suffix of us starting at position i , $0 \leq i \leq m$. The length of us_i is $m - i$, so us_m is the empty string and $us_0 = us$. Define vs_j , $0 \leq j \leq n$, similarly. Store $ed(us_i, vs_j)$ in $cost[i, j]$, $0 \leq i \leq m$ and $0 \leq j \leq n$. The required minimum cost is $cost[0, 0] = ed(us_0, vs_0) = ed(us, vs)$.

The program is shown in Figure 7.22. Define $(i', j') < (i, j)$ if $i' + j' > i + j$ and all variables are between respective array bounds. Even though this ordering is not explicitly used in the program, it is used in invariant I shown in the program as an assertion.

The computation time and space are $\mathcal{O}(m \times n)$. The space can be reduced considerably by computing along the diagonals and keeping a pointer to a neighboring entry in the matrix to compute the edit sequence, exactly as in the longest common subsequence problem.

7.7 Exercises

1. What is $\max x y + \min x y$?
2. Define a function over three positive integer arguments, p, q and r , that determines if p, q and r are the lengths of the sides of a triangle. Recall from geometry that the combined length of any two sides of a triangle is greater than the third.

Hint First, define function `max3` that returns the maximum of 3 integer arguments.

Solution

```
max3 p q r = max p (max q r)
triangle p q r = (p+q+r) > (2 * (max3 p q r))
```

3. Define a function that returns $-1, 0$ or $+1$, depending on whether the argument integer is negative, zero or positive.
4. Compute x^y using only multiplication. Employ the same strategy as in `quickMlt` of Section 7.2.5.4 (page 389).
5. A point in two dimensions is a pair of coordinates, each a floating point number. A line is given by the pair of its (distinct) end points. Define function `parallel` over two lines whose value is `True` iff the lines are parallel. So,

```
parallel ((3.0,5.0),(3.0,8.0)) ((3.0,5.0),(3.0,7.0)) is True and
parallel ((3.0,5.0),(4.0,8.0)) ((4.0,5.0),(3.0,7.0)) is False.
```

Hint Recall from coordinate geometry that two lines are parallel iff their slopes are equal, and the slope of a line is given by the difference of the y -coordinates of its two points divided by the difference of their x -coordinates. In order to avoid division by 0, avoid division altogether.

Solution This program is due to Jeff Chang, a former student in my class.

```
parallel ((a,b),(c,d)) ((u,v),(x,y)) =
  (d-b) * (x-u) == (y-v) * (c-a)
```

6. Given two lists of equal length, define boolean function `hamming` whose value is true iff the lists differ in exactly one position in the values of their corresponding elements. For example, (`hamming "abc" "abd"`) is `True`, and (`hamming "abc" "abc"`) is `False`.

Solution

```
hamming [ ] [ ] = False
hamming (x:xs) (y:ys) =
  ((x /= y) && (xs == ys)) || ((x == y) && (hamming xs ys))
```

Richard Hamming introduced the notion of distance between bit strings, the number of positions in which they differ. Function `hamming` determines if they are at unit distance.

7. Develop a function, similar to `merge` of Section 7.2.8.3 (page 400), that computes the intersection of two sorted lists.

Solution

```
intersect xs [ ] = [ ]
intersect [ ] ys = [ ]
intersect (x:xs) (y:ys)
  | x < y    = intersect xs (y:ys)
  | x > y    = intersect (x:xs) ys
  | x == y   = x: (intersect xs ys)
```

8. The definitions of `andl` and `orl` in Section 7.2.9.2 (page 401) look wasteful. If a scanned element of a list is `False`, it is not necessary to scan the list any further to compute the value of `andl`. Modify `foldr` accordingly. Assume that operator f is associative, and it has a zero element z so that $f\ x\ z = f\ z\ x = z$ for all x . Thus, 0 is the zero for $*$, for instance. But not every operator has a zero, for example $+$.
9. Modify the algorithm for removing unnecessary white spaces, as described in Section 7.2.10 (page 404), so that a trailing white space, as in `Mary-had-a-little-lamb-`, is also removed.

Solution Add the following two lines to the definitions.

```
first ['-'] = []
next ['-'] = []
```

10. Consider the naive algorithm for computation of Fibonacci numbers in Section 7.2.5.2 (page 388). Show that during the computation of `fib n`, $n > 1$, `fib m` for any m , $0 < m < n$, is called F_{n+1-m} times.

Hint Apply induction on $m + n$. Pay special attention to $m = n - 1$ and $m = n - 2$.

11. Prove each of the following statements for function `count` defined in Section 7.3.2 (page 407):

- (a) $\text{count}(2^n) = 1$, for all $n, n \geq 0$,
- (b) $\text{count}(2^n - 1) = n$, for all $n, n > 0$,
- (c) $\text{count}(2^n + 1) = 2$, for all $n, n > 0$.

12. I proved in Section 7.3.4.1 (page 411) that $\text{filter } p (\text{filter } p \text{ } xs) = \text{filter } p \text{ } xs$, for all p and xs . Prove the more general result:

$\text{filter } p (\text{filter } q \text{ } xs) = \text{filter } (p \wedge q) \text{ } xs$, for all p, q and xs .

Hint For the inductive case, consider four different predicates that may be true of the first element of the list: $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$ and $\neg p \wedge \neg q$.

13. The algorithm for maximum segment sum given in Figure 7.8 (page 419) computes only the length of the segment having the maximum sum. Modify the algorithm to compute the maximum segment itself.

Hint Index the elements of a list starting at 0 for the rightmost element. Let the value of `mss xs` be a pair of indices (i, j) denoting the boundaries of the maximum segment in xs ; according to the usual convention the segment includes i but excludes j , so that (i, i) is an empty segment. Modify the definition of `mss`.

14. Write a function that identifies a longest substring of a given string of text that does not contain the character 'e'. Therefore, given the string "The quick brown fox jumps over the lazy dog" the desired substring is "quick brown fox jumps ov".

Hint Assign value 1 to each letter other than 'e' in the text and $-\infty$ to letter 'e'. Find the segment having the maximum sum.

15. (Gray code)

- (a) Gray code for $n, n \geq 0$, is the list of all n -bit strings so that the adjacent strings differ in exactly one bit position; the first and last strings are considered adjacent, as in wraparound adjacency. For example, with $n = 3$,

`["000", "001", "011", "010", "110", "111", "101", "100"]`

is a possible Gray code. The Gray code for $n = 0$ is the list containing the empty string, `[""]`. Devise a function for computing Gray code for any argument $n, n \geq 0$.

Suggestion Let `gray(n)` be the list of strings for any n . Then

`gray (n+1) = append0 (gray n) ++ append1 (reverse (gray n))`

where `append0` appends '0' to the beginning of every string of its argument list of strings, and `append1` appends '1'. Prove the correctness of this strategy. You can implement `append0` as `map ('0':)`, and `append1` similarly.

- (b) Generalize the definition of `gray`, as suggested in part(a), to `gray2` so that

`gray2 n = (gray n, reverse (gray n)).`

Define `gray2` so that list reversal is completely avoided.

Solution In the recursive case, adapt this code.

`gray2 (n+1) = append0 x ++ append1 y`

where

`(x,y) = gray2 n`

- (c) Here is another, roundabout, procedure to generate Gray code. Start with an n -bit string of all zeros. Number the bits 1 through n , from lower to higher bits. Solve the Towers of Hanoi problem for n , and whenever disk i is moved, flip the i^{th} bit of your number and record it. Show that all 2^n strings of n -bit are recorded exactly once in a Gray code fashion.
16. (a) Define a function to list the values in a given binary search tree (bst) in increasing order.
- (b) A balanced binary search tree (bbst) is a bst that meets the following additional constraint: root of a subtree of size $2.s + 1, s \geq 0$, has left and right subtrees of size s each, and root of a subtree of size $2.s$ has subtrees that differ in size by 1. Define a function to construct a bbst from a list of values that are in increasing order.

Hint for the second part Define a more general function `baltree` where `baltree xs n` has `xs`, a non-empty list of values, in increasing order, and `n` a number between 1 and the length of `xs`. The value of `baltree xs n` is a tuple `(bstree, rest)` where `bstree` is a `bbst` constructed from the first `n` elements of `xs` and `rest` is the remaining suffix of `xs`.

Solution

```

baltree (x:xs) 1 = (Leaf x, xs)
baltree (x:y:xs) 2 = (Node(Leaf x, y, Null), xs)
baltree xs n      = (Node(left, v, right), zs)
                    where
                        n' = n `div` 2
                        (left, v:ys) = baltree xs n'
                        (right, zs)  = baltree ys (n
                                                - n')

makebst xs          = fst (baltree xs (length xs))

```

17. (Subexpression identification) Given two expressions e and f in the forms of trees, determine if e is a subexpression of f . Base your solution on the ideas of tree isomorphism described in Section 7.5.2 (page 428).

Hint It will help to add size information to each subtree of f ; this is done in linear time by applying a simple recursive function. Both trees should then be normalized. Next, recursively search a subtree of f only if the size of the subtree is at least the size of e .

18. (Full binary tree enumeration) Enumerate *all* full binary trees of n nodes, for a given positive integer n . Code a dynamic programming solution.

Hint A full binary tree of n nodes has s nodes in its left subtree and t in its right, where $s > 0$, $t > 0$, and $n = 1 + s + t$. Thus, the solutions for s and t can be computed once and used for computations with higher values. Also see the solution to Exercise 19.

The number of full binary trees with $k + 1$ leaves, or equivalently k internal nodes, is the k^{th} *Catalan* number.

19. Given is a circle with an even number of points on its periphery; points are numbered consecutively with natural numbers starting at 0 and going counterclockwise along the circle. A *configuration* is a set of straight lines that connect pairs of points, where (1) each point is on exactly one line and (2) the lines do not intersect. Enumerate all the configurations. Develop a dynamic programming solution.

A personal note This is the first program I wrote for which I got paid; I was then an undergraduate. I did not know recursion nor dynamic programming, so I struggled with coding the solution using the dominant programming language of the day, Fortran.

Hint This is actually the same problem as in Exercise 18; instead of full binary trees you are asked to enumerate all trees, as explained below.

A configuration is a set of pairs of points, each pair denoting a line. A configuration over points $0..n$ is possible only if n is even. In any configuration point 0 is connected to k where k is odd; this line divides the circle into two non-overlapping regions each having an even number of points, one with points numbered $1..k$ and the other with points $k+1..n$. So, this configuration can be represented as a binary tree where the root is $(0, k)$, the left subtree a configuration over points $1..k$ and the right subtree over $k+1..n$. Different values of k yield different configurations.

Solution Let $conf(i, j)$ be the set of configurations for a circle whose points are numbered $i..j$, where $j - i$ is even. For even n :

$$\begin{aligned} & conf(0, n) \\ &= (\cup k : 0 < k < n, \text{ odd}(k) : \text{set of configurations where 0 and } k \\ & \quad \text{are connected}). \end{aligned}$$

The definition gives the empty set for $conf(0, 0)$, as expected. A configuration where 0 and k are connected is a set that includes the pair $(0, k)$ and a configuration from $conf(1, k-1)$ and one from $conf(k+1, n)$ as subsets. Hence, the set of configurations where 0 and k are connected is:

$$\{left \cup \{(0, k)\} \cup right \mid left \in conf(1, k-1), right \in conf(k+1, n)\}.$$

$$\begin{aligned} & conf(0, n) \\ &= (\cup k : 0 < k < n, \text{ odd}(k) : \\ & \quad \{left \cup \{(0, k)\} \cup right \mid left \in conf(1, k-1), right \in conf(k+1, n)\} \\ & \quad) \end{aligned}$$

You do not need a recursive definition for $conf(i, j)$ because it can be computed by adding i to every number in every pair in every configuration in $conf(0, j-i)$. Develop a dynamic programming solution by computing $conf(0, n)$ in increasing order of n .

20. (Longest increasing subsequence) An *increasing subsequence* of a sequence of integers is one in which the values are increasing along the subsequence. It is required to find a longest increasing subsequence of a given sequence of distinct numbers. I sketch two solutions using dynamic programming that take

$\mathcal{O}(n^2)$ time. Then I sketch an $\mathcal{O}(n \log n)$ algorithm derived using the extension heuristic of Section 7.4.3.1 (page 417).

In the first two solutions, I compute only the length of the longest increasing subsequence; you can easily modify it to compute the subsequence itself.

A $\mathcal{O}(n^2)$ algorithm Store the given sequence as an array $A[0..n]$ of integers. Let d_i be the length of the longest increasing subsequence that has $A[i]$ as its last element. Observe that d_i is non-zero because the sequence that includes only $A[i]$ is an increasing subsequence. Compute d_i in increasing order of i ; take $d_0 = 1$. For $i > 0$, the last item in the increasing subsequence is $A[i]$ and the remaining prefix, if non-empty, is an increasing subsequence ending at j , where $j < i$. Hence, find the largest d_j such that $j < i$ and $A[i] > A[j]$, then $d_i = 1 + d_j$. If there is no such j , then the prefix is empty and $d_i = 1$. The computation of j takes $\mathcal{O}(n)$ for each i , so this is an $\mathcal{O}(n^2)$ algorithm.

Another $\mathcal{O}(n^2)$ algorithm Sort A in increasing order to get another sequence B . Show that the set of common subsequences of A and B are the increasing subsequences of A . Next, find the longest common subsequence of A and B using the lcs algorithm in Section 7.6.4 (page 439).

A $\mathcal{O}(n \log n)$ algorithm This algorithm is developed using the extension heuristic of Section 7.4.3.1 (page 417). In fact, the algorithm appears in Misra [1978] as an example of application of the heuristic.

Let S be the given sequence of positive distinct numbers; assume that the first element of S is 0 in order to simplify the algorithm description. Let lis_i be an increasing subsequence of length i , $i > 0$, in the given sequence. There may be many increasing subsequences of length i . Then the one whose last element (call it the *tip*) is as small as possible is the “best longest increasing subsequence” of length i , or $blis_i$. And $blis_m$, where m is the largest possible value, is a longest increasing subsequence.

- (1) Let v_i be the value of the tip of $blis_i$. Prove that $v_i < v_{i+1}$, for all i , $1 \leq i < m$.
- (2) Compute $blis_i$, for all possible i , using the extension heuristic. Initially, $m = 1$ and $blis_m = \langle 0 \rangle$. For $S \leftarrow x$:

```

if  $x > v_m$ 
  then { extend  $blis_m$  to get  $blis_{m+1}$  }
     $blis_{m+1}, m, v_{m+1} := blis_m \leftarrow x, m + 1, x$ 
  else { modify  $blis_{i+1}$  where  $v_i < x < v_{i+1}$  }
    replace the tip of  $blis_{i+1}$  by  $x$ ;
     $v_{i+1} := x$ 
endif

```


(3) Prove the invariant that each $blis_i$ meets the condition for being the best longest increasing subsequence of length i . Therefore, $blis_m$ is a longest increasing subsequence at every stage.

(4) Observe that the index i that satisfies $v_i < x < v_{i+1}$ in the **else** clause can be computed using binary search, from the condition in (1). So, each extension takes $\mathcal{O}(\log m)$ time and the entire algorithm $\mathcal{O}(n \log n)$ time. Propose an efficient data structure for storing all $blis_i$ so that binary search may be applied on their tips.

21. Given are two lists of numbers of arbitrary lengths. There are two players, call them **True** and **False** for simplicity (no aspersion intended), who alternately remove the first number from either list, starting with a turn for **True**. If one of the lists becomes empty, players remove the numbers alternately from the other list until it becomes empty. The game ends when both lists are empty.

Each player has a *purse*, the sum of the numbers she has picked. The goal for each player is to maximize her purse by the end of the game. For example, given the lists $[10, 7]$ and $[2, 12]$, player **True** can at best have a purse of 12, so **False** will have 19 in her purse. To see this, if **True** removes 10, then **False** removes 7, forcing **True** to remove 2 to gain a score of 12 and **False** to score 19. And, if **True** initially removes 2, then **False** removes 12, again forcing the same scores for both players.

An optimal play is where both players make the best possible move at each point assuming that the opponent will always make the best possible move. Devise a polynomial algorithm in the lengths of the lists to compute the purses under optimal play.

Hint First, devise a top-down solution. I show one below. Prove that the solution has exponential time complexity in the lengths of the lists. Derive a dynamic programming solution from the top-down solution similar to the solution for the lcs problem.

A top-down solution In Figure 7.23, function `play` simulates all possible plays for both players and chooses the best strategy for each. It has three arguments, `b xs ys` denoting that player `b` has the next move and `xs ys` are the remaining lists. The value of `play b xs ys` is a pair of values, the purses of `b` and `not b` in an optimal play when the starting lists are `xs` and `ys`. Below, `purseb` and `purseb'` denote the purses of `b` and `not b`. The first two clauses handle the situation when one list is empty. In the last clause both lists are non-empty. So both scenarios, where `b` takes the next value from `xs` and from `ys`, are simulated with optimal play starting with the resulting lists.

Optimal 2-player game

```

play b [ ] [ ] = (0,0)
play b (x:xs) [ ] = (purseb+x, pursedb')
    where (purseb',purseb) = play (not b) xs [ ]

play b [ ] (y:ys) = (purseb+y, pursedb')
    where (purseb',purseb) = play (not b) [ ] ys

play b (x:xs) (y:ys) =
    if pursedb1+x > pursedb2+y
    then (purseb1+x, pursedb'1) -- choose x
    else (purseb2+y, pursedb'2) -- choose y
    where
        (purseb'1,purseb1)
        = play (not b) xs (y:ys)

        (purseb'2,purseb2)
        = play (not b) (x:xs) ys

```

Figure 7.23

A sample output is:

```

*Main> play True [10,7] [2,12]
(12,19)

```

The reader may enhance this solution by creating the sequence of moves that leads to the optimal purses.

7.A

7.A.1

Appendices

Towers of Hanoi: Simulating an Iterative Solution

An iterative solution for the tower of Hanoi compares the top disks of two specific pegs in each step and moves the smaller disk from one peg to the other. Assume that a peg that has no disk holds a disk numbered $n+1$. The comparisons are performed in a specific order. If n is even then the comparisons on pairs of pegs are done in order in the following sequence $\langle (S, I), (D, S), (I, D) \rangle$ repeatedly, and

if n is odd the order is $\langle (S, D), (I, S), (D, I) \rangle$. This fact can be proved from the recursive solution `tower1`. Observe that the given triple for even n can be encoded more simply as (S, I, D) , and for odd n as (S, D, I) , where each move compares the top disks of the first two pegs in the triple and then rotates the triple once to the right for the next move.

I simulate this iterative solution by a recursive program. The state of computation at any point are the contents of the pegs and the sequence of moves to be performed. Function `loop` has three arguments, each argument has the form (p, xs) , where p is a peg, S, D or I , and xs the list of disks stacked on it. The order of the pegs in the arguments is such that the first two pegs are compared in the next move. For example, given the arguments $(p, x:xs) (q, y:ys) (r, zs)$, compare the top disks of pegs p and q , that is, x and y . If $x < y$ the next move is (p, x, q) , that is, move x from p to q . The remaining computation is performed on the arguments $(r, zs) (p, xs) (q, x:y:ys)$, the result of moving x from p to q and rotating the sequence of arguments once to the right. Similar reasoning applies for $x > y$.

The next issue is how to terminate the computation. The computation terminates when the two pegs that are compared have no disks on them; then all the disks are on the remaining peg, D . To simplify this check, we adopt yet another idea common in imperative programming, *sentinel*. A sentinel is usually a data item that does not belong in the original problem description but introduced solely to simplify programming. For the tower of Hanoi problem, introduce three sentinel disks, each with value $n + 1$, and put them initially at the bottom of each peg. The existence of the sentinel does not affect the moves at all. However, if the two disks in a comparison are found to be equal, they must be $n + 1$ because no other pair of disks are equal.

Definition of `tower3 n` in Figure 7.24 initializes the states of the pegs, using `enum (n+1)`, which is the list $[1, 2, \dots, n+1]$. Each call to `loop` performs a single move, updates the state, and calls itself. The clause $x == y = []$ in `loop` is justified by the termination considerations.

Observe that the calls to `loop` are tail-recursive. So, most implementations will replace recursive computation by iteration.

Further improvements Further analysis of the given program yields additional improvements. First, I show that in every alternate move, starting with the first move, the first argument of `loop` has the form $(p, 1: _)$, that is, disk 1 is at the top of the peg that appears as the first argument. I will use this observation to eliminate comparison of disks in alternate moves.

Initially, the assertion holds because the first argument is $(S, \text{enum}(n+1))$ and the head of `enum(n+1)` is 1. Next, assume inductively that the result holds before any

Iterative tower of Hanoi

```

enum 0 = []
enum n = enum(n-1) ++ [n]

loop (p,x:xs) (q,y:ys) (r,zs)
  | x == y = []
  | x < y  = (p,x,q): (loop (r,zs) (p,xs)      (q,x:y:ys) )
  | x > y  = (q,y,p): (loop (r,zs) (p,y:x:xs) (q,ys)      )

tower3 n
  | even(n) = loop (S,enum(n+1)) (I,[n+1]) (D,[n+1])
  | odd(n)  = loop (S,enum(n+1)) (D,[n+1]) (I,[n+1])

```

Figure 7.24

move numbered $2n+1$, $n > 0$ (the first move is numbered 1). I show the arguments of `loop` before three successive moves starting at move number $2n+1$. Before move number $2n+1$, the state, as given in line (1), has arbitrary disk placements except that the first peg has disk 1, inductively. Since 1 is the smallest disk, the state after this move has disk 1 on top of peg q and the three arguments rotated once to the right, as shown in line (2). Move number $2n+2$ moves a disk between r and p and rotates the arguments once to the right resulting in the configuration in line (3). That is, before move number $2(n+1)+1$ disk 1 is at the top of the peg that appears as the first argument.

Before move $2n+1$: $(p, 1:_) (q, _) (r, _)$ (1)

Before move $2n+2$: $(r, _) (p, _) (q, 1:_)$ (2)

Before move $2n+3$: $(q, 1:_) (r, _) (p, _)$ (3)

I exploit this observation in Figure 7.25 by using two different functions, `loopodd` and `loopeven`, to treat the odd and even numbered moves. For odd moves no comparison is needed, as shown in its definition. Further, note that each odd move involves disk 1, therefore $x == y$ is bound to fail; so the clause $x == y = []$ can also be eliminated. Functions `loopodd` and `loopeven` call each other for the next move. I reuse function `enum` from Figure 7.24.

It is possible to eliminate `loopodd` from the definition; replace each call to it by the expression in its definition, modifying the call arguments suitably.

Improvements to tower3

```

loopodd (p,x:xs) (q,y:ys) (r,zs) =
  (p,x,q): (loopeven (r,zs) (p,xs) (q,x:y:ys) )

loopeven (p,x:xs) (q,y:ys) (r,zs)
  | x == y = []
  | x < y = (p,x,q): (loopodd (r,zs) (p,xs) (q,x:y:ys) )
  | x > y = (q,y,p): (loopodd (r,zs) (p,y:x:xs) (q,ys) )

tower4 n
  | even(n) = loopodd (S,enum(n+1)) (I,[n+1]) (D,[n+1])
  | odd(n) = loopodd (S,enum(n+1)) (D,[n+1]) (I,[n+1])

```

Figure 7.25**7.A.2 Reducing Space Usage in the *lcs* Computation**

I propose two schemes for reducing the storage space for the program that computes the longest common subsequence in Figure 7.19 of Section 7.6.4 (page 440): (1) store only the part of the matrix that is needed for subsequent computation, and (2) store the information needed to recreate the subsequence at any $lcsmat[i,j]$ instead of the entire subsequence itself.

For (1), to compute $lcsmat[i,j]$ we need $lcsmat[i+1,j]$, $lcsmat[i,j+1]$ and $lcsmat[i+1,j+1]$. Let $diag_t$ be the diagonal of $lcsmat$ in which the sum of the row and column index of each element is t . Then the computation of $diag_t$ requires values only from $diag_{t+1}$ and $diag_{t+2}$. Therefore, the computation for $lcsmat[0,0]$ can proceed by storing values for only three diagonals, taking up at most $3 \times \max(m,n)$ space. It is possible to reduce the storage requirements even further because the entries of $diag_t$ that remain to be computed and the parts of $diag_{t+1}$ and $diag_{t+2}$ that are needed for their computation can fit within $2 \times \max(m,n)$ storage.

For (2), observe that $lcsmat[i,j]$, for $0 \leq i < m$ and $0 \leq j < n$, is either (i) $us[i] ++ lcsmat[i+1,j+1]$, or (ii) $lcsmat[i,j+1]$, or (iii) $lcsmat[i+1,j]$. So, instead of storing the actual subsequence, store just the *length* of the subsequence, that is, $length(lcs(us_i, vs_j))$, at $lcsmat[i,j]$. This allows computation of all the entries of $lcsmat$, and upon completion of this phase the desired sequence, $lcs(us, vs)$, can be reconstructed as follows.

Longest common subsequence

```

{ Initialize }
{ Corresponding to lcs us [ ] = [ ] }
  for  $i \in \text{rev}(0..m+1)$  do  $lcsmat[i,n] := 0$  endfor ;
{ Corresponding to lcs [ ] vs = [ ] }
  for  $j \in \text{rev}(0..n+1)$  do  $lcsmat[m,j] := 0$  endfor ;

{ Computation proper }
  for  $i \in \text{rev}(0..m)$  do
    for  $j \in \text{rev}(0..n)$  do
      if  $us[i] = vs[j]$  then  $lcsmat[i,j] := 1 + lcsmat[i+1,j+1]$ 
      else {  $us[i] \neq vs[j]$  }
        if  $lcsmat[i,j+1] > lcsmat[i+1,j]$ 
          then  $lcsmat[i,j] := lcsmat[i,j+1]$ 
          else  $lcsmat[i,j] := lcsmat[i+1,j]$ 
        endif
      endif
    endfor
  endfor ;

{ Compute  $lcs(us, vs)$  by starting at  $lcsmat[0,0]$  }
 $i, j := 0, 0$ ;
 $dseq := []$ ;
{  $lcs(us, vs) = dseq ++ lcs(us_i, vs_j)$  }
while  $i \neq m \wedge j \neq n$  do
{ Invariant:  $lcs(us, vs) = dseq ++ lcs(us_i, vs_j)$  }
  if  $us[i] = vs[j]$  then
     $dseq := dseq ++ "us[i]"$  ;  $(i, j) := (i+1, j+1)$ 
  else {  $us[i] \neq vs[j]$  }
    if  $lcsmat[i,j+1] > lcsmat[i+1,j]$ 
      then  $(i, j) := (i, j+1)$  else  $(i, j) := (i+1, j)$ 
    endif
  endif
enddo
{  $lcs(us, vs) = dseq ++ lcs(us_i, vs_j), i = m \vee j = n$  }
{  $lcs(us, vs) = dseq ++ []$  }
{  $lcs(us, vs) = dseq$  }

```

Figure 7.26 Space-efficient lcs computation.

Let $dseq$ be a sequence and (i, j) index to an entry in $lcsmat$. Maintain the invariant: $lcs(us, vs) = dseq ++ lcs(us_i, vs_j)$.

Initially, the invariant is established by setting $i, j, dseq = 0, 0, []$. Finally, $i = m$ or $j = n$, so $lcs(us_i, vs_j) = []$ and $lcs(us, vs) = dseq$. During the computation, if $us[i] = vs[j]$, then $lcs(us, vs)$ includes the symbol $us[i]$ and the remaining part of $lcs(us, vs)$ is $lcs(us_{i+1}, vs_{j+1})$, that is, $lcs(us, vs) = dseq ++ "us[i]" ++ lcs(us_{i+1}, vs_{j+1})$. So, set $dseq := dseq ++ "us[i]"$ and $(i, j) := (i + 1, j + 1)$ to preserve the invariant. If $us[i] \neq vs[j]$ and $lcs(us_i, vs_{j+1})$ is longer than $lcs(us_{i+1}, vs_j)$, that is, $lcsmat[i, j + 1] > lcsmat[i + 1, j]$, then $lcs(us, vs) = dseq ++ lcs(us_i, vs_{j+1})$. Similarly, if $us[i] \neq vs[j]$ and $lcsmat[i, j + 1] < lcsmat[i + 1, j]$, then $lcs(us, vs) = dseq ++ lcs(us_{i+1}, vs_j)$. So, set $(i, j) := (i, j + 1)$ in the first case and $(i, j) := (i + 1, j)$ in the second case.

The program is shown in Figure 7.26.

Parallel Recursion

8.1 Parallelism and Recursion

This book is primarily about sequential programs, their formal descriptions and analysis. This chapter is the sole exception; it is about a specific type of parallel programs, often called *synchronous* or *data* parallel programs. The purpose of this chapter is not to teach parallel programming techniques in detail but show that formal methods, induction and recursion in particular, that have been developed earlier in this book are applicable in this new domain. I draw upon functional programming ideas and notations from Chapter 7 and enhance them with a novel data structure, called *powerlist*, that is suitable for describing a class of synchronous parallel algorithms, called *hypercubic* algorithms.

Role of recursion and induction in parallel computations We in computer science are extremely fortunate that recursion is applicable for descriptions and induction for analysis of sequential programs. In fact, they are indispensable tools in programming. Sequentiality is inherent in mathematics. The Peano axioms define natural numbers in a sequential fashion, starting with 0 and defining each number as the successor of the previous one. One can see its counterpart in Haskell; the fundamental data structure, list, is either empty, corresponding to 0, or one formed by appending a single element to a smaller list. Naturally, induction is the major tool for analyzing programs on lists.

There is no counterpart of parallelism in classical mathematics because traditional mathematics is not concerned about mechanisms of computations, sequential or parallel. Any possible parallel structure is sequentialized (or *serialized*) so that traditional mathematical tools, including induction, can be used for its analysis. Therefore, most parallel recursive algorithms are typically described iteratively, one parallel step at a time. The mathematical properties of the algorithms are rarely evident from these descriptions.

Multi-dimensional matrix I study multi-dimensional matrices where the length of each dimension is a power of two. Such a structure can be halved along any

dimension that has length of more than one, yielding two similar structures. So, recursive decomposition and computation at all but the lowest level is possible.

Hypercube A hypercube is a machine with a multi-dimensional matrix structure. There is a processor, or node, at each element of a hypercube. A *neighbor* of a node is any other node whose index differs in exactly one dimension. Neighbors are physically connected so that they can exchange data. A problem is solved on a hypercube by initially storing data at each node and having each node carry out similar computations and data exchanges in synchronous parallel steps. Final steps, which may not be synchronous, accumulate the result in one or more nodes. Typically, the number of nodes in a hypercube is a power of 2.

Structure of the chapter Many important synchronous parallel algorithms—Fast Fourier Transform, routing and permutation, Batcher sorting schemes, solving tridiagonal linear systems by odd-even reduction, prefix sum algorithms—are conveniently formulated in a recursive fashion. The network structures on which parallel algorithms are typically implemented—butterfly, sorting networks, hypercube, complete binary tree—are, also, recursive in nature. Descriptions of these algorithms and the network structures are surprisingly simple using powerlists. Simple algebraic properties of powerlists permit us to deduce properties of these algorithms by employing structural induction.

8.2 Powerlist

I describe powerlists in the context of functional programming. The descriptions are in the style of Haskell from Chapter 7, though powerlist is not implemented as a part of Haskell. Data items of Haskell, which are predefined or defined by the user, are called *scalars* in this chapter and a Haskell list a *linear* list to distinguish it from powerlist.

A powerlist is a list of data items much like a linear list but with its own constructors. A powerlist is finite and its length is a power of 2; so the smallest powerlist has just one item. As in a linear list all items of a powerlist are of the same type. A powerlist's items are enclosed within angular brackets, as in $\langle 2\ 3 \rangle$, to distinguish it from linear lists.¹ The items are separated by either white spaces or commas. A powerlist is a data item itself, so it can be embedded within other kinds of data structures and its items can be any data structure including powerlists.

1. Unfortunately, the notation clashes with that for sequences. In this chapter, I rarely use sequences other than powerlists, so there should be no confusion.

8.2.1 Powerlist Constructors

Two powerlists are *similar* if they have the same length and their elements are either scalars of the same type or similar powerlists.

A powerlist is either (1) $\langle s \rangle$, a singleton list of scalar s , (2) $\langle p \rangle$, a singleton list of powerlist p , or formed by (3) concatenating two similar powerlists p and q , written as $p \mid q$, or (4) interleaving two similar powerlists u and v , written as $u \bowtie v$, by taking each element alternately from u and v starting with u .

The constructors $\mid$ and $\bowtie$ are called *tie* and *zip*. The length of a singleton powerlist is 2^0 , and the length of a powerlist doubles by applying either constructor; so, the length of any powerlist is a power of 2. Examples of powerlists are shown below.

$\langle \rangle$ not a powerlist
 $\langle 2 \rangle$ singleton powerlist
 $\langle 0 \rangle \mid \langle 1 \rangle$ = $\langle 0 \ 1 \rangle$
 $\langle 0 \ 1 \rangle \mid \langle 2 \ 3 \rangle$ = $\langle 0 \ 1 \ 2 \ 3 \rangle$
 $\langle 0 \ 1 \rangle \bowtie \langle 2 \ 3 \rangle$ = $\langle 0 \ 2 \ 1 \ 3 \rangle$

The following powerlists have more elaborate structures.

$\langle \langle 2 \rangle \rangle$ nested powerlist
 $\langle [] \rangle$ powerlist containing the empty linear list
 $\langle \text{"00"}, \text{"01"}, \text{"10"}, \text{"11"} \rangle$ powerlist of strings, which are linear lists
 $\langle \langle 0 \ 4 \rangle \langle 1 \ 5 \rangle \langle 2 \ 6 \rangle \langle 3 \ 7 \rangle \rangle$ matrix $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$ stored column-wise
 $\langle \langle 0 \ 1 \ 2 \ 3 \rangle \langle 4 \ 5 \ 6 \ 7 \rangle \rangle$ same matrix stored row-wise
 $\langle \langle \langle a \rangle \langle b \rangle \rangle \langle \langle c \rangle \langle d \rangle \rangle \rangle$ balanced binary tree of Figure 8.1.

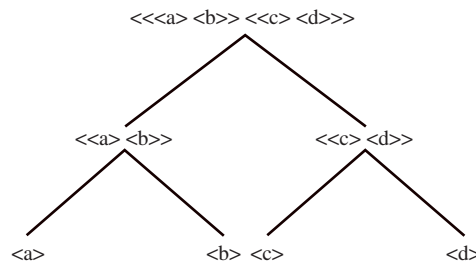


Figure 8.1 Balanced binary tree. Each tree is a powerlist of both subtrees.

Indices in a powerlist Associate index i with the element at position i in a powerlist, where i starts at 0. In $p \mid q$, p is the sublist of elements whose highest bit in

the binary expansion of the index is 0 and q 's elements have 1 as their highest bit. Dually, in $u \bowtie v$ the lowest bit of the index is 0 for the elements of u and 1 for v .

Discussion The base case of a powerlist is a singleton list, not an empty list, corresponding to the length 2^0 . Empty lists (or, equivalent data structures) do not arise in data parallel programming. For instance, in matrix algorithms the base case is a 1×1 matrix rather than an empty matrix, Fourier transform is defined for a singleton list (not the empty list) and the smallest hypercube has one node.

The recursive definition of a powerlist says that a powerlist is either of the form $u \bowtie v$ or $u | v$. In fact, every non-singleton powerlist can be written in either form in a unique manner (see Laws in Section 8.2.3). It is peculiar to have two different ways of constructing the same powerlist, using tie or zip. This flexibility is essential because many parallel algorithms exploit both forms of construction, the Fast Fourier Transform being the most prominent (see Section 8.5.3, page 475).

The powerlist structure permits recursive divide and conquer. Further, since each division creates powerlists of the same size and structure, they can be processed in parallel if there are enough available processors. Square matrices are often processed by quartering them. I show in Section 8.6 (page 492) how quartering, or quadrupling, can be expressed in this theory.

8.2.2 Simple Function Definitions over Powerlists

Just as functions over linear lists are typically defined by case analysis over empty and non-empty lists, functions over powerlists are defined separately for singleton and non-singleton lists. I show a number of functions over single-dimensional powerlists in the following pages. Multi-dimensional powerlists are treated in Section 8.6 (page 492), where the constructors $|$ and $\bowtie$ are generalized to apply to specific dimensions.

Henceforth, I adopt the convention that the name of the powerlist function corresponding to a known linear list function is the same with symbol “ p ” appended at the end; so the powerlist counterpart of the linear list function *fun* is *funp*.

Length functions over powerlists Function *lenp* computes the length of a powerlist, corresponding to function *len* for a linear list. Since the length of a powerlist is a power of 2, the logarithmic length, *lgp*, is sometimes a more useful measure.

$$\begin{array}{l|l} \text{lenp } \langle x \rangle = 1 & \text{lgp } \langle x \rangle = 0 \\ \text{lenp } (p | q) = 2 \times (\text{lenp } p) & \text{lgp } (p | q) = 1 + (\text{lgp } p) \end{array}$$

Search for a value Function *find* returns *true* if v is an element of the argument powerlist, *false* otherwise.

$$\begin{array}{lcl} \text{find } v \langle x \rangle & = & x = v \\ \text{find } v (p | q) & = & (\text{find } v p) \vee (\text{find } v q) \end{array}$$

Pairwise sum of adjacent values Function *sum2* has a non-singleton powerlist *p* of integers as its argument. It returns a pair of values, (*evensum*, *oddsum*) where *evensum* is the sum of the elements with even indices and *oddsum* of odd indices. It uses *sum*, a function that returns the sum of the elements of a powerlist, as a helper function.

$$\begin{aligned} \text{sum } \langle x \rangle &= x \\ \text{sum } (p \mid q) &= (\text{sum } p) + (\text{sum } q) \\ \text{sum2 } (p \bowtie q) &= (\text{sum } p, \text{sum } q) \end{aligned}$$

Reverse a powerlist Function *revp* reverses its argument powerlist.

$$\begin{aligned} \text{revp } \langle x \rangle &= \langle x \rangle \\ \text{revp } (p \mid q) &= (\text{revp } q) \mid (\text{revp } p) \end{aligned}$$

I have used $\mid$ for deconstruction in this definition. I could have used $\bowtie$ and defined *revp* in the recursive case by $\text{revp}(p \bowtie q) = (\text{revp } q) \bowtie (\text{revp } p)$. I show in Section 8.4.1 (page 466) that the two definitions of *revp* are equivalent.

8.2.3 Laws

The following properties of powerlists are easily proved. Below, $\langle x \rangle$ and $\langle y \rangle$ are singleton powerlists, and *p*, *q*, *u* and *v* arbitrary powerlists.

$$\text{L0. } \langle x \rangle \mid \langle y \rangle = \langle x \rangle \bowtie \langle y \rangle.$$

L1. (Dual Deconstruction)

For a non-singleton powerlist *P* there are powerlists *p*, *q*, *u* and *v* such that

$$P = p \mid q \text{ and } P = u \bowtie v.$$

L2. (Unique Deconstruction)

$$\begin{aligned} \langle \langle x \rangle = \langle y \rangle \rangle &\equiv (x = y) \\ (p \mid q = u \mid v) &\equiv (p = u \wedge q = v) \\ (p \bowtie q = u \bowtie v) &\equiv (p = u \wedge q = v) \end{aligned}$$

L3. (Distributivity of the constructors)

$$(p \mid q) \bowtie (u \mid v) = (p \bowtie u) \mid (q \bowtie v)$$

These laws can be derived by suitably defining *tie* and *zip*, using the standard functions from the linear list theory. One possible strategy is to define *tie* as the concatenation of two equal length lists and then use L0 and L3 as the definition of *zip*; next, derive L1 and L2.

Law L0 is often used in proving base cases of algebraic identities. Laws L1 and L2 permit us to uniquely deconstruct a non-singleton powerlist using either $\mid$

or $\bowtie$. The distributivity law, Law L3, is crucial. It is the only law relating both constructors. Hence, it is invariably applied in proofs by structural induction where both constructors play a role.

Conventions about binding power, pattern matching Constructors $|$ and $\bowtie$ have the same binding power, and their binding power is lower than that of function application.

I adopt a more general style of pattern matching that is not admissible in Haskell; I allow simple arithmetic expressions such as $n + 1$, $2 \times k$ and $2 \times k + 1$ for a pattern. This extension simplifies descriptions of several algorithms.

8.2.4 Non-standard Powerlists

To compute a function in parallel on a linear list of *arbitrary* length (1) pad the linear list with artificial elements so that its length becomes a power of 2, (2) create a powerlist from the elements of the linear list and apply a relevant powerlist function on it, and (3) compute the desired function value from the powerlist function value, by applying another function.

For example, to compute the sum of the elements of a linear list, pad it with zeroes so that the list length is a power of 2, create a powerlist from its elements, and then apply a sum function on this powerlist. The choice of the artificial elements, 0s in the case of the sum function, was easy. To search for a value v using the *find* function of Section 8.2.2 (page 462) artificial element is some value different from v . To sort a linear list pad it with elements which are smaller (or larger) than all the elements in the list; they can be discarded after sorting.

I show a function, *lin2pow*, to create a powerlist out of a linear list *xs* whose length is a power of 2. I use function *split*, defined in Section 7.2.8.1 (page 399), that partitions the elements of the argument list into two lists, elements with even index go in the first list and with odd index in the second list (list elements are indexed starting at 0).

$$\begin{aligned} \text{lin2pow } [x] &= \langle x \rangle \\ \text{lin2pow } xs &= \text{lin2pow}(ys) \bowtie \text{lin2pow}(zs) \\ \text{where } (ys, zs) &= \text{split } xs \end{aligned}$$

8.3 Pointwise Function Application

Call a function a *scalar* if all its arguments are scalar. Similarly, a scalar operator is a binary operator over scalar arguments.

Scalar function f is applied *pointwise* to powerlist p means that f is applied to each element of p to form a powerlist; notationally, this application is shown as $(f \ p)$. Similarly, a scalar binary operator $\oplus$ applied pointwise to similar powerlists

p and q , and written as $p \oplus q$, means that $\oplus$ is applied to the corresponding elements of p and q to form a single powerlist. Thus, given that function *inc* increases its integer argument by 1, $\text{inc}\langle 2\ 3 \rangle = \langle 3\ 4 \rangle$, and $\langle 2\ 3 \rangle \times \langle 0\ 4 \rangle = \langle 0\ 12 \rangle$.

Formally, for powerlists of scalars

$$\begin{aligned} f \langle x \rangle &= \langle f x \rangle \\ f (p | q) &= (f p) | (f q) \\ f (p \bowtie q) &= (f p) \bowtie (f q), \text{ which can be derived.} \end{aligned}$$

And,

$$\begin{aligned} \langle x \rangle \oplus \langle y \rangle &= \langle x \oplus y \rangle \\ (p | q) \oplus (u | v) &= (p \oplus u) | (q \oplus v) \\ (p \bowtie q) \oplus (u \bowtie v) &= (p \oplus u) \bowtie (q \oplus v), \text{ which can be derived.} \end{aligned}$$

Both of these forms of function application are instances of *map* operation defined on linear lists (of Section 7.2.9.3, page 403) generalized for powerlists. Define *mapp* and *mapp2* as shown below. Then $(f\ p)$ is $(\text{mapp } f\ p)$ and $(p \oplus q)$ is $(\text{mapp2 } \oplus\ p\ q)$.

$$\begin{aligned} \text{mapp } f \langle x \rangle &= \langle f x \rangle \\ \text{mapp } f (p | q) &= (\text{mapp } f\ p) | (\text{mapp } f\ q) \\ \text{mapp2 } \oplus \langle x \rangle \langle y \rangle &= \langle x \oplus y \rangle \\ \text{mapp2 } \oplus (p | q) (u | v) &= (\text{mapp2 } \oplus\ p\ u) | (\text{mapp2 } \oplus\ q\ v) \end{aligned}$$

It is easy to show that *mapp* and *mapp2* distribute over both tie and zip:

$$\begin{aligned} \text{mapp } f (p | q) &= (\text{mapp } f\ p) | (\text{mapp } f\ q), \\ \text{mapp } f (p \bowtie q) &= (\text{mapp } f\ p) \bowtie (\text{mapp } f\ q), \\ \text{mapp2 } \oplus (p | q) (u | v) &= (\text{mapp2 } \oplus\ p\ u) | (\text{mapp2 } \oplus\ q\ v), \\ \text{mapp2 } \oplus (p \bowtie q) (u \bowtie v) &= (\text{mapp2 } \oplus\ p\ u) \bowtie (\text{mapp2 } \oplus\ q\ v). \end{aligned}$$

Note Pointwise application is valid only for a scalar function or operator; so operators $|$ and $\bowtie$ can not be applied in place of $\oplus$.

8.3.1 Examples of Scalar Function Application

Scalar function *rev* (defined in Section 7.4.3.4, page 420) reverses a linear list. Applied to a powerlist of strings, where a string is a linear list in Haskell, it reverses each linear list, for example, $\text{rev}\langle \langle "00" "01" \rangle \langle "10" "11" \rangle \rangle$ is $\langle \langle "00" "10" \rangle \langle "01" "11" \rangle \rangle$. In contrast, *revp* reverses a powerlist, not its individual elements.

Figure 8.2 shows the application of *rev* and *revp* to a powerlist of strings p , where p is the powerlist of indices of the corresponding positions written as a string in binary. Observe that $\text{rev}(\text{revp } p) = \text{revp}(\text{rev } p)$.

p	$=$	$\langle$	"000"	"001"	"010"	"011"	"100"	"101"	"110"	"111"	$\rangle$
$rev\ p$	$=$	$\langle$	"000"	"100"	"010"	"110"	"001"	"101"	"011"	"111"	$\rangle$
$revp\ p$	$=$	$\langle$	"111"	"110"	"101"	"100"	"011"	"010"	"001"	"000"	$\rangle$
$rev\ (revp\ p)$	$=$	$\langle$	"111"	"011"	"101"	"001"	"110"	"010"	"100"	"000"	$\rangle$
$revp\ (rev\ p)$	$=$	$\langle$	"111"	"011"	"101"	"001"	"110"	"010"	"100"	"000"	$\rangle$

Figure 8.2 rev and $revp$ applied to a powerlist of strings that are indices.

8.3.2 Powerlist of Indices

Define $(idx\ n)$, for $n \geq 0$, to be the powerlist of all n -bit strings in order, so it is the powerlist of indices of length 2^n . So, $(idx\ 0) = \langle "" \rangle$, $(idx\ 1) = \langle "0" "1" \rangle$, $(idx\ 2) = \langle "00" "01" "10" "11" \rangle$, and $(idx\ 3)$ is the powerlist p in Figure 8.2.

Define $(cidx\ n)$ as the powerlist obtained from $(idx\ n)$ by complementing each bit of each index, so $(cidx\ 2) = \langle "11" "10" "01" "00" \rangle$, and $(cidx\ 3)$ is $(revp\ p)$ in Figure 8.2. Observe that $cidx$ gives the sequence of indices in reverse order of idx .

Below ('0' :) and $(++\ "0")$ are scalar functions that append '0' to the front and back of a string, respectively.

$$\begin{aligned} idx\ 0 &= \langle "" \rangle \\ idx\ (n+1) &= ((\text{'0' :})(idx\ n)) \mid ((\text{'1' :})(idx\ n)) \end{aligned}$$

$$\begin{aligned} cidx\ 0 &= \langle "" \rangle \\ cidx\ (n+1) &= ((\text{'1' :})(cidx\ n)) \mid ((\text{'0' :})(cidx\ n)) \end{aligned}$$

It is easy to show that

$$\begin{aligned} idx\ (n+1) &= (++\ "0")(idx\ n) \bowtie (++\ "1")(idx\ n) \\ cidx\ (n+1) &= (++\ "1")(cidx\ n) \bowtie (++\ "0")(cidx\ n) \end{aligned}$$

8.4 Proofs

8.4.1 Inductive Measures on Powerlists

Most proofs on powerlists are by structural induction on its length or its logarithmic length. Multi-dimensional powerlists require other measures, which I propose in Section 8.6 (page 492). Here I show one typical proof:

$$revp(p \bowtie q) = (revp\ q) \bowtie (revp\ p).$$

- p and q are singleton lists, $\langle x \rangle$ and $\langle y \rangle$, respectively:

$$\begin{aligned} & revp(\langle x \rangle \bowtie \langle y \rangle) \\ = & \{ \langle x \rangle \bowtie \langle y \rangle = \langle x \rangle \mid \langle y \rangle, \text{law L0} \} \\ & revp(\langle x \rangle \mid \langle y \rangle) \\ = & \{ \text{definition of } revp \} \end{aligned}$$

$$\begin{aligned}
& (revp \langle y \rangle) | (revp \langle x \rangle) \\
= & \{ \text{definition of } revp \} \\
& \langle y \rangle | \langle x \rangle \\
= & \{ \langle y \rangle | \langle x \rangle = \langle y \rangle \bowtie \langle x \rangle, \text{ law L0} \} \\
& \langle y \rangle \bowtie \langle x \rangle \\
= & \{ \text{definition of } revp \} \\
& revp \langle y \rangle \bowtie revp \langle x \rangle
\end{aligned}$$

• $p = u | v, q = r | s$:

$$\begin{aligned}
& revp(p \bowtie q) \\
= & \{ p = u | v, q = r | s \} \\
& revp((u | v) \bowtie (r | s)) \\
= & \{ \text{distributivity of } | \text{ and } \bowtie, \text{ law L3} \} \\
& revp((u \bowtie r) | (v \bowtie s)) \\
= & \{ \text{definition of } revp \} \\
& revp(v \bowtie s) | revp(u \bowtie r) \\
= & \{ \text{induction} \} \\
& ((revp s) \bowtie (revp v)) | ((revp r) \bowtie (revp u)) \\
= & \{ \text{distributivity of } | \text{ and } \bowtie, \text{ law L3} \} \\
& ((revp s) | (revp r)) \bowtie ((revp v) | (revp u)) \\
= & \{ \text{definition of } revp \} \\
& revp(r | s) \bowtie revp(u | v) \\
= & \{ p = u | v, q = r | s \} \\
& (revp q) \bowtie (revp p)
\end{aligned}$$

8.4.2 Permutation Functions

Functions that permute the items of their argument powerlists are important in many hypercubic computations. We have seen permutation function *revp* that reverses a powerlist. I show a number of permutation functions in this section and in later sections.

A permutation of n items is defined by a bijection Π over indices $0..n$. Application of Π to powerlist p creates the powerlist $\Pi \triangleright p$ by storing the element at position $\Pi(i)$ in p at position i in $\Pi \triangleright p$. So, for $p = \langle x_0 x_1 \dots x_{2^n-1} \rangle$, $\Pi \triangleright p = \langle x_{\Pi(0)} x_{\Pi(1)} \dots x_{\Pi(2^n-1)} \rangle$. A permutation function f implements Π if $(f p) = (\Pi \triangleright p)$, for all p .

Observe that sorting a list permutes its elements, but sort is not a permutation function because the permutation achieved is dependent on data values in the list. A permutation function permutes the elements solely based on their positions in the list.

I show a number of permutation functions for 1-dimensional powerlists and in Section 8.6 (page 492) for multi-dimensional powerlists. I develop a proof theory for permutation functions.

8.4.3 Examples of Permutation Functions

8.4.3.1 Rotate

Function rr rotates a powerlist to the right by one; thus, $rr\langle a\ b\ c\ d\rangle = \langle d\ a\ b\ c\rangle$. Function rl rotates to the left: $rl\langle a\ b\ c\ d\rangle = \langle b\ c\ d\ a\rangle$.

$$\begin{array}{lcl} rr\langle x\rangle = \langle x\rangle & \parallel & rl\langle x\rangle = \langle x\rangle \\ rr(u \bowtie v) = (rr\ v) \bowtie u & \parallel & rl(u \bowtie v) = v \bowtie (rl\ u) \end{array}$$

There does not seem to be any simple definition of rr or rl using $|$ as the deconstruction operator. It is easy to show, using structural induction, that rr and rl are inverses.

Function grr , below, rotates powerlist p to the right by a given amount k . This applies for positive, negative and 0 values of k . For k greater than 2^n , the length of p , the rotation is effectively by $k \bmod 2^n$. Instead of applying k single rotations, grr rotates its two component powerlists by about $k/2$.

$$\begin{array}{lcl} grr\ k\ \langle x\rangle & = & \langle x\rangle \\ grr\ (2 \times k)\ (u \bowtie v) & = & (grr\ k\ u) \bowtie (grr\ k\ v) \\ grr\ (2 \times k + 1)\ (u \bowtie v) & = & (grr\ (k + 1)\ v) \bowtie (grr\ k\ u) \end{array}$$

8.4.3.2 Swap

Any permutation of a list can be achieved by a suitable number of swaps of adjacent elements (see Section 3.4.1, page 101). I show a function that swaps any pair of wraparound adjacent elements of a powerlist. Exercise 6 (page 501) asks for a function that swaps any two elements, not necessarily adjacent.

Write $(swap\ i\ j\ p)$, where p is a powerlist of length 2^n , $n > 0$, i and j are wraparound adjacent indices in p , so $j = i + 1 \bmod 2^n$, and both i and j are given as n -bit strings. The effect of the call is to swap the elements at positions i and j . Thus, $(swap\ "0"\ "1"\ list2)$ swaps the two elements of a 2-element list, $(swap\ "11"\ "00"\ list4)$ swaps the two end elements of a 4-element list, and $(swap\ "0111"\ "1000"\ list16)$ swaps the middle two elements of a 16-element list.

The function definition appears in Figure 8.3. The base case of a two-element list merely swaps its two elements. For a longer list, with four or more elements, suppose the indices have the same leading bit, 0 or 1. Then the items to be swapped belong in the same half of the argument powerlist, and they are swapped with a recursive call on that half.

$$\begin{aligned}
\text{swap } - - \langle x y \rangle &= \langle y x \rangle \\
\text{swap } (x : xs) (y : ys) (u \mid v) &= \\
&\mid (x, y) = ('0', '0') \rightarrow (\text{swap } xs \ ys \ u) \mid v \\
&\mid (x, y) = ('1', '1') \rightarrow u \mid (\text{swap } xs \ ys \ v) \\
&\mid (x, y) = ('0', '1') \rightarrow (ul \mid ur') \mid (vl' \mid vr) \\
&\text{where} \\
&\quad u = (ul \mid ur) \\
&\quad v = (vl \mid vr) \\
&\quad (vl' \mid ur') = \text{swap } xs \ ys \ (vl \mid ur) \\
&\mid (x, y) = ('1', '0') \rightarrow (ul' \mid ur) \mid (vl \mid vr') \\
&\text{where} \\
&\quad u = (ul \mid ur) \\
&\quad v = (vl \mid vr) \\
&\quad (ul' \mid vr') = \text{swap } xs \ ys \ (ul \mid vr)
\end{aligned}$$

Figure 8.3 Swap adjacent items in a powerlist.

If the leading bits of the indices are different, the items to be swapped, x and y , are in different halves. Since x and y are wraparound adjacent, either they are the middle two elements or the two end elements of the list. Partition the argument powerlist into four parts in sequence, so the powerlist is $(ul \mid ur) \mid (vl \mid vr)$.

- x and y are the middle two elements: Then x is the last element of ur and y the first element of vl . Then i and j have 0 and 1 as their leading bits, respectively. So, y is the first element and x the last element of $(vl \mid ur)$. Their indices in this powerlist are i' and j' , where i' is the same as i with its leading bit removed, and similarly for j' . Swap x and y recursively in $(vl \mid ur)$ to get $(vl' \mid ur')$. The required result is the powerlist $((ul \mid ur') \mid (vl' \mid vr))$.

- x and y are the two end elements: So, x is the first element of ul and y the last element of vr . Using similar argument as in Case (1), swap x and y recursively in $(ul \mid vr)$ to get $(ul' \mid vr')$. The required result is the powerlist $((ul' \mid ur) \mid (vl \mid vr'))$.

Each of the recursive calls is made to a powerlist of half the size, so the number of calls is exactly n and swap takes $\mathcal{O}(n)$ time, the logarithmic length of the argument powerlist. Swapping any two items, not necessarily adjacent, also takes $\mathcal{O}(n)$ time (see Exercise 6, page 501). Clearly, this is much worse than swapping two items of an array in $\mathcal{O}(1)$ time, but this is a necessary cost in a hypercubic computation because the items to be swapped may be as far as n in the hypercube.

8.4.3.3 Permute Index

Any bijection h , mapping indices to indices as binary strings, defines a permutation of the powerlist; the element with index i is moved to the position where it has index $(h\ i)$. Function rs (for *right shuffle*) rotates an index in binary to the right by one position and ls to the left by one position. Figure 8.4 shows the effects of index rotations on a powerlist p . For instance, the element g that is at index 110 in the original list p is moved to index 011 after applying rs .

indices	(000	001	010	011	100	101	110	111)
p	$\langle a$	b	c	d	e	f	g	$h \rangle$
$rs\ p$	$\langle a$	c	e	g	b	d	f	$h \rangle$
$ls\ p$	$\langle a$	e	b	f	c	g	d	$h \rangle$

Figure 8.4 Permutation functions rs , ls .

The definition of rs , given below, may be understood as follows. The effect of rotating an index to the right is that the lowest bit of an index becomes the highest bit; therefore, if rs is applied to $u \bowtie v$, the elements of u , that is, those having 0 as the lowest bit, will occupy the first half of the resulting powerlist (because their indices have 0 as the highest bit, after rotation); similarly, v will occupy the second half. The argument for ls is analogous. It is easy to see that rs and ls are inverses. The definitions use both tie and zip.

$$\begin{array}{l} rs\langle x \rangle = \langle x \rangle \\ rs(u \bowtie v) = u \mid v \end{array} \quad \parallel \quad \begin{array}{l} ls\langle x \rangle = \langle x \rangle \\ ls(u \mid v) = u \bowtie v \end{array}$$

8.4.3.4 Inversion

Function inv moves the element of index i to $(rev\ i)$, as shown below on a powerlist of length 8.

$$\begin{array}{cccccccc} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\ inv\langle & a & b & c & d & e & f & g & h \rangle \\ = & \langle & a & e & c & g & b & f & d & h \rangle \end{array}$$

The definition of inv uses both tie and zip.

$$\begin{array}{lcl} inv\langle x \rangle & = & \langle x \rangle \\ inv(p \mid q) & = & (inv\ p) \bowtie (inv\ q) \end{array}$$

This function arises in a variety of contexts. In particular, inv is used to permute the output of a Fast Fourier Transform network into the correct order.

The identity $inv(p \bowtie q) = (inv\ p) \mid (inv\ q)$ holds. Its proof is standard, using structural induction, along the same lines as the proof for $revp$ in Section 8.4.1

(page 466) (see Appendix 8.A.1, page 504, for a proof). It is also easy to establish that $inv(inv P) = P$.

The powerlist constructors and inv share a few properties of the operators of boolean algebra. Taking $|$ and $\bowtie$ to be $\wedge$ and $\vee$, inv may be regarded as negation because it satisfies De Morgan's law. Constructors tie and zip satisfy the distributivity laws but not many other laws.

8.4.3.5 Gray Code

I introduced Gray code in Exercise 15 of Chapter 7 (page 448). To recapitulate, a Gray code for n , $n \geq 0$, is a sequence of all n -bit strings in which adjacent strings differ in exactly one bit position; the first and last strings in the sequence are considered adjacent, so there is wraparound adjacency among the strings. For example, with $n = 3$,

["000", "001", "011", "010", "110", "111", "101", "100"]

is a Gray code. The Gray code for $n = 0$ is the sequence containing the empty string, [""].

In the context of powerlists, define function $gray\ n$, $n \geq 0$, to be the corresponding powerlist of strings, using the same strategy as suggested in Exercise 15 of Chapter 7 (page 448): append '0' to $(gray\ n)$ and concatenate it with the reverse of the sequence obtained by appending '1' to $(gray\ n)$.

$$\begin{aligned} gray\ 0 &= \langle "" \rangle \\ gray\ (n + 1) &= ('0' :)(gray\ n) | revp(('1' :)(gray\ n)) \end{aligned}$$

For later use of this function, I devise a small variation; the parameter n is replaced by a powerlist, and the goal is to compute a gray code of the same length as the powerlist.

$$\begin{aligned} gr\ \langle x \rangle &= \langle "" \rangle \\ gr\ (p | q) &= ('0' :)(gr\ p) | revp(('1' :)(gr\ q)) \end{aligned}$$

Function pgr Function pgr permutes any powerlist p according to $(gr\ p)$, that is, each item of p moves to the position determined by the index at the corresponding position of $(gr\ p)$. Thus, for $p = \langle a\ b\ c\ d \rangle$, $(gr\ p) = \langle "00" "01" "11" "10" \rangle$, so $pgr\ \langle a\ b\ c\ d \rangle = \langle a\ b\ d\ c \rangle$.

$$\begin{aligned} pgr\ \langle x \rangle &= \langle x \rangle \\ pgr\ (p | q) &= (pgr\ p) | pgr(revp\ q) \end{aligned}$$

You are asked to prove the correctness of pgr in Exercise 7 (page 501).

8.4.3.6 Permutations Distribute over Scalar Functions

A permutation function rearranges data items of its argument powerlist according to some permutation but not modify the items themselves. A scalar function may modify the data items but not rearrange them. Therefore, we expect a permutation function to distribute over a scalar function. That is, for permutation function f and scalar function h , first applying h to powerlist p and then permuting the resulting powerlist has the same effect as first permuting p and then applying h , and similarly for scalar binary operator $\oplus$:

$$\begin{aligned} f(h\ p) &= h(f\ p) \\ f(p \oplus q) &= (f\ p) \oplus (f\ q) \end{aligned}$$

We expect similarly that the application of permutation Π and scalar function h also distribute:

$$\begin{aligned} \Pi \triangleright (h\ p) &= h(\Pi \triangleright p) \\ \Pi \triangleright (p \oplus q) &= (\Pi \triangleright p) \oplus (\Pi \triangleright q) \end{aligned}$$

We can prove these identities by using induction on the length of the argument list and properties of scalar functions.

8.4.4 Correctness of Permutation Functions

In order to prove the correctness of a permutation function, we first need a specification that describes a unique property of the function. It is not sufficient to specify, for example, that $revp(revp\ p) = p$ because many other functions, such as the identity function, also meet such a specification.

Suppose we can prove that $revp\langle 0\ 1\ 2\ 3 \rangle = \langle 3\ 2\ 1\ 0 \rangle$. Then I claim that $revp$ reverses every powerlist of four items. More formally,

Permutation principle A permutation function that permutes *any* powerlist of *distinct items* correctly, permutes all powerlists of the same length correctly. ■

This principle is proved easily. Let f be a permutation function that implements a permutation Π , so for any powerlist of distinct items $(f\ p) = (\Pi \triangleright p)$. The principle claims that $(f\ q) = (\Pi \triangleright q)$ for any q of the same length as p , not necessarily of distinct items. Call p a *witness* for f implementing Π .

To justify this principle, let h be a scalar function such that $h(p_i) = q_i$, for all i , so $(h\ p) = q$.

$$\begin{aligned} & f\ q \\ = & \{q = (h\ p)\} \\ & f(h\ p) \\ = & \{f\ \text{distributes over scalar } h\} \end{aligned}$$

$$\begin{aligned}
& h(f\ p) \\
= & \{(f\ p) = (\Pi \triangleright p)\} \\
& h(\Pi \triangleright p) \\
= & \{\Pi \text{ distributes over scalar } h\} \\
& \Pi \triangleright (h\ p) \\
= & \{(h\ p) = q\} \\
& \Pi \triangleright q
\end{aligned}$$

The permutation principle provides a powerful proof (and testing) strategy for permutation functions. To prove that f is a permutation function, choose a family of witnesses of all possible lengths and show that a given permutation function permutes all of them correctly. The witnesses are typically powerlists of indices.

Function $revp$ can be proved using the powerlists of indices as witnesses. The specification of $revp$ is that $revp\langle 0\ 1\ \dots\ 2^n - 1 \rangle = \langle 2^n - 1\ \dots\ 1\ 0 \rangle$, for all n . It is, however, easier to work with indices in the form of bit-strings. I use idx , introduced in Section 8.2.2 (page 462), and show that $revp\ (idx\ n)$ is the reverse of $(idx\ n)$, which is $(cidx\ n)$, for all n , $n \geq 0$. I prove $revp$ next and inv in Appendix 8.A.1 (page 504).

8.4.4.1 Proof of $revp$

I show $revp(idx\ n) = (cidx\ n)$, for all n , $n \geq 0$, by induction on n .

• $n = 0$:

$$\begin{aligned}
& revp(idx\ 0) \\
= & \{\text{from definition of } idx\} \\
& revp\langle "" \rangle \\
= & \{\text{definition of } revp\} \\
& \langle "" \rangle \\
= & \{\text{from definition of } cidx\} \\
& cidx\ 0
\end{aligned}$$

• $n + 1$, $n \geq 0$:

$$\begin{aligned}
& revp(idx\ (n + 1)) \\
= & \{\text{definition of } idx\} \\
& revp((('0' :)(idx\ n)) \mid (('1' :)(idx\ n))) \\
= & \{\text{definition of } revp\} \\
& revp(('1' :)(idx\ n)) \mid revp(('0' :)(idx\ n)) \\
= & \{revp \text{ distributes over scalar function } ('1' :) \text{ and } ('0' :)\} \\
& ('1' :)(revp(idx\ n)) \mid ('0' :)(revp(idx\ n)) \\
= & \{\text{induction on } n\}
\end{aligned}$$

$$\begin{aligned}
& ('1' :)(cidx\ n) \mid ('0' :)(cidx\ n) \\
= & \{\text{definition of } cidx\} \\
& cidx(n + 1)
\end{aligned}$$

8.5 Advanced Examples Using Powerlists

I show the use of powerlists in several well-known problems, such as the Fast Fourier Transform (Section 8.5.3, page 475), Batcher sorting schemes (Section 8.5.4, page 479), and prefix sum (Section 8.5.5, page 485).

8.5.1 *fold*

I defined a higher order function *foldr* for linear lists in Section 7.2.9 (page 400). For a binary prefix operator *f* whose unit element is *u*, (*foldr f u xs*) has value *u* if *xs* is empty, otherwise the value is the result of applying *f* to the elements of *xs* in sequence. I define a similar function, *foldp*, over powerlists. There is no empty powerlist, so there is no need for the unit element. Below, $\oplus$ is an associative binary infix operator.

$$\begin{aligned}
foldp \oplus \langle x \rangle &= x \\
foldp \oplus (p \mid q) &= (foldp \oplus p) \oplus (foldp \oplus q)
\end{aligned}$$

The examples used for linear lists have their counterparts for powerlists:

$$\begin{aligned}
sump &= foldp \ + \\
prodp &= foldp \ \times \\
andp &= foldp \ \wedge \\
orp &= foldp \ \vee \\
xorp &= foldp \ \neq
\end{aligned}$$

The definition $foldp \oplus (p \mid q) = (foldp \oplus p) \oplus (foldp \oplus q)$ suggests a similar identity with *zip* as the constructor:

$$foldp \oplus (p \bowtie q) = (foldp \oplus p) \oplus (foldp \oplus q).$$

This identity holds only if $\oplus$ is commutative as well as associative.

8.5.2 Polynomial

We studied polynomial evaluation in Section 7.4.1 (page 414). Here we adapt those procedures for powerlists.

A polynomial with coefficients p_j , $0 \leq j < 2^n$, where $n \geq 0$, may be represented by powerlist *p* whose j^{th} element is p_j . The polynomial value at some real or complex value ω is $(+j : 0 \leq j < 2^n : p_j \times \omega^j)$.

The polynomial value can be written in terms of two polynomial values, one containing the coefficients with even indices and the other with odd indices:

$$\begin{aligned} & (+i : 0 \leq i < 2^{n-1} : p_{2,i} \times \omega^{2,i}) + (+i : 0 \leq i < 2^{n-1} : p_{2,i+1} \times \omega^{2,i+1}) \\ & = (+i : 0 \leq i < 2^{n-1} : p_{2,i} \times \omega^{2,i}) + \omega \times (+i : 0 \leq i < 2^{n-1} : p_{2,i+1} \times \omega^{2,i}). \end{aligned}$$

Function *poly* evaluates the expression for a non-singleton powerlist at a specified value:

$$\text{poly } (p \bowtie q) \omega = \text{poly } p \omega^2 + \omega \times (\text{poly } q \omega^2).$$

Each of the component polynomials can be evaluated in parallel and recursively.

I generalize this definition to permit the second argument to be a powerlist *w* of several numbers, not just a single number ω . The polynomial is evaluated at each element of *w* and the evaluation returns a powerlist of the same length as *w*, the polynomial value at each point of *w*.

Below, function *ep* is given as an infix operator over powerlists *p* and *w*, of possibly unequal lengths, where *p* represents a polynomial and *w* the points where *p* is to be evaluated. For number *r*, (*r*×) is a scalar function that multiplies its operand by *r*; applied to a powerlist, it multiplies each element by *r*.

$$\begin{aligned} \langle x \rangle \text{ep } \langle _ \rangle &= \langle x \rangle \\ (p \bowtie q) \text{ep } \langle r \rangle &= (p \text{ep } r^2) + (r \times) (q \text{ep } r^2) \\ p \text{ep } (u \mid v) &= (p \text{ep } u) \mid (p \text{ep } v) \end{aligned}$$

8.5.3 Fast Fourier Transform

Fast Fourier Transform (FFT) [see [Cooley and Tukey 1965](#)] is an outstanding example of an algorithm that reshaped an entire area of engineering, in this case digital signal processing. Fourier transform is applied to a function of time and it yields a function of frequency, from which a great deal of useful information about the underlying physical phenomenon can be obtained. We study the Discrete Fourier Transform in which a finite sequence of values, call them *time-values* because they are usually obtained by sampling some continuous function over time, are transformed into *frequency-values* that is a finite sequence of the same length. The time-values may be regarded as the coefficients of a polynomial and the frequency-values as the polynomial values at some specific points. Prior to Fast Fourier Transform, this computation took $\mathcal{O}(N^2)$ sequential time for a polynomial of degree *N*; after Fast Fourier Transform it takes $\mathcal{O}(N \log N)$ sequential time, and $\mathcal{O}(\log N)$ parallel time on a hypercube with *N* processors, a significant improvement.

8.5.3.1 Problem Description

The Fourier transform, FT , of powerlist p of length N is a powerlist of length N . Treat p as a polynomial as in Section 8.5.2; then $FT\ p$ is the value of p at the N primitive roots of 1 (unity), which are defined as follows. Let ω be $e^{-2\pi i/N}$, where e is the base of natural logarithm and i the unit imaginary number. For each k , $0 \leq k < N$, ω^k is a primitive N^{th} root of 1, and the powerlist $\langle \omega^0 \ \omega^1 \ \dots \ \omega^{N-1} \rangle$ is the sequence of N primitive roots. Write $(roots\ p)$ to denote $\langle \omega^0 \ \omega^1 \ \dots \ \omega^{N-1} \rangle$, where the length of p is N ; function $roots$ ignores the values of the items in p , just using its length. For similar powerlists p and q , $(roots\ p) = (roots\ q)$.

Observe that $roots\langle x \rangle = 1$ and $roots(p \bowtie q) = (roots\ p) \bowtie \omega \times (roots\ q)$, where the length of p is N and ω is $e^{-2\pi i/N}$.

The Fourier transform of p is given by function FT where $FT\ p = p\ ep\ (roots\ p)$, and ep is the polynomial evaluation function defined in Section 8.5.2. Fast Fourier Transform exploits the structure of $(roots\ p)$, that it is not an arbitrary powerlist of values but the primitive roots of unity, to achieve its efficiency.

The Fourier transform is reversible. Inverse of the Fourier transform maps the frequency-values to the original time values. I develop that algorithm in Section 8.5.3.3 (page 478).

8.5.3.2 Derivation of the Fast Fourier Transform Algorithm

I use the following properties of $roots$ for the derivation of FFT .

$$\begin{aligned} roots^2(p \bowtie q) &= (roots\ p) \mid (roots\ q) && \text{(roots1)} \\ roots(p \bowtie q) &= u \mid (-u), \text{ for some powerlist } u && \text{(roots2)} \end{aligned}$$

Equation (roots1) can be proved from the definition of $roots$ using the fact that $\omega^{2 \times N} = 1$, where N is the length of p (and q). Equation (roots2) says that the right half of $roots(p \bowtie q)$ is the negation of its left half. This is because each element in the right half is the same as the corresponding element in the left half multiplied by ω^N , which is -1 because $\omega = (2 \times N)^{\text{th}}$ root of 1, so $\omega^N = -1$.

Next, I derive identities for FT for the singleton and non-singleton cases.

- Base case, singleton powerlist:

$$\begin{aligned} &FT\langle x \rangle \\ &= \{\text{definition of } FT\} \\ &\quad x\ ep\ (roots\langle x \rangle) \\ &= \{\text{roots}\langle x \rangle \text{ is a singleton. Use the definition of } ep\} \\ &\quad \langle x \rangle \end{aligned}$$

- Inductive case, non-singleton powerlist:

$$\begin{aligned}
& FT(p \bowtie q) \\
= & \{\text{definition of } FT\} \\
& (p \bowtie q) \text{ ep roots}(p \bowtie q) \\
= & \{\text{definition of ep}\} \\
& (p \text{ ep roots}^2(p \bowtie q)) + (\text{roots}(p \bowtie q) \times (q \text{ ep roots}^2(p \bowtie q))) \\
= & \{\text{equation (roots1): } \text{roots}^2(p \bowtie q) = (\text{roots } p) \mid (\text{roots } q)\} \\
& p \text{ ep } ((\text{roots } p) \mid (\text{roots } q)) \\
& + (\text{roots}(p \bowtie q) \times (q \text{ ep } ((\text{roots } p) \mid (\text{roots } q)))) \\
= & \{\text{distribute each ep over its second argument}\} \\
& ((p \text{ ep } (\text{roots } p)) \mid (p \text{ ep } (\text{roots } q))) \\
& + (\text{roots}(p \bowtie q) \times ((q \text{ ep } (\text{roots } p)) \mid (q \text{ ep } (\text{roots } q)))) \\
= & \{(\text{roots } p) = (\text{roots } q).\} \\
& \text{Inductively, } p \text{ ep } (\text{roots } p) = (FT \text{ } p), \text{ } q \text{ ep } (\text{roots } q) = (FT \text{ } q). \} \\
& ((FT \text{ } p) \mid (FT \text{ } p)) + \text{roots}(p \bowtie q) \times ((FT \text{ } q) \mid (FT \text{ } q)) \\
= & \{\text{use } P, Q \text{ for } FT \text{ } p, FT \text{ } q, \text{ and} \\
& u \mid (-u) \text{ for } \text{roots}(p \bowtie q) \text{ from equation (roots2)}\} \\
& (P \mid P) + ((u \mid -u) \times (Q \mid Q)) \\
= & \{| \text{ and } \times \text{ in the second term distribute}\} \\
& (P \mid P) + ((u \times Q) \mid (-u \times Q)) \\
= & \{| \text{ and } + \text{ distribute}\} \\
& (P + u \times Q) \mid (P - u \times Q)
\end{aligned}$$

Equations for FFT Collect the equations for FT to define FFT , the Fast Fourier Transform.

$$\begin{aligned}
FFT\langle x \rangle &= \langle x \rangle \\
FFT(p \bowtie q) &= (P + u \times Q) \mid (P - u \times Q) \\
\text{where } P &= FFT \text{ } p \\
Q &= FFT \text{ } q \\
(u \mid _) &= \text{roots}(p \bowtie q)
\end{aligned}$$

The compactness of this description of FFT is in striking contrast to the usual descriptions. The compactness can be attributed to the use of recursion and the avoidance of explicit indexing of the elements by employing $\mid$ and $\bowtie$. Function FFT illustrates the need for including both $\mid$ and $\bowtie$ as constructors for powerlists. Other functions that employ both $\mid$ and $\bowtie$ is IFT , the inverse of FFT that is described below, and inv of Section 8.4.3.4 (page 470).

Complexity It is clear that $FFT(p \bowtie q)$ can be computed from $(FFT \text{ } p)$ and $(FFT \text{ } q)$ in $O(N)$ sequential steps or $O(1)$ parallel steps using $O(N)$ processors (u can be

computed in parallel), where N is the length of p . Therefore, $FFT(p \bowtie q)$ can be computed in $O(N \log N)$ sequential steps or $O(\log N)$ parallel steps using $O(N)$ processors.

There are a few minor optimizations in the computation of FFT . Precompute (roots r) for all powerlists r up to the length of the argument list of FFT . And for computing $FFT(p \bowtie q)$ as $(P + u \times Q) \mid (P - u \times Q)$, compute $u \times Q$ just once and use it in both terms.

8.5.3.3 Inverse Fourier Transform

I derive the inverse of the Fourier Transform, IFT , from that of the FFT by pattern matching. For a singleton powerlist $\langle x \rangle$:

$$\begin{aligned} IFT\langle x \rangle &= \{ \langle x \rangle = FFT\langle x \rangle \} \\ &= \{ IFT(FFT\langle x \rangle) \} \\ &= \{ IFT, FFT \text{ are inverses} \} \\ &\quad \langle x \rangle \end{aligned}$$

To compute IFT for non-singleton powerlists, let $IFT(r \mid s) = p \bowtie q$ in the unknowns p and q . I derive the values of p and q next. This form of deconstruction allows us to solve the equations easily. Taking FFT of both sides

$$\begin{aligned} FFT(IFT(r \mid s)) &= FFT(p \bowtie q), \text{ or} \\ (r \mid s) &= FFT(p \bowtie q), \text{ since } IFT \text{ and } FFT \text{ are inverses.} \end{aligned}$$

Replace $FFT(p \bowtie q)$ in the right side by its definition from Section 8.5.3.2:

$$\begin{aligned} r \mid s &= (P + u \times Q) \mid (P - u \times Q) \\ P &= FFT\ p \\ Q &= FFT\ q \\ (u \mid _) &= roots(p \bowtie q) \end{aligned}$$

These equations are easily solved for the unknowns P , Q , u , p and q . Using the law of unique deconstruction, L2, deduce from the first equation that $r = P + u \times Q$ and $s = P - u \times Q$, so $P = (r + s)/2$ and $Q = ((r - s)/2)/u$; here $(/2)$ divides each element of the given powerlist by 2 and $(/u)$ divides each element of the dividend powerlist by the corresponding element of u . Further, since p and r are of the same length, define u using r instead of p . The solutions of these equations yield the following definition for IFT .

Equations for IFT

$$\begin{aligned} IFT\langle x \rangle &= \langle x \rangle \\ IFT(r \mid s) &= p \bowtie q \end{aligned}$$

where

$$\begin{aligned}
 P &= (r + s)/2 \\
 Q &= ((r - s)/2)/u \\
 p &= IFT\ P \\
 q &= IFT\ Q \\
 (u \mid _) &= roots(r \bowtie s)
 \end{aligned}$$

As in *FFT*, the definition of *IFT* includes both constructors, $\mid$ and $\bowtie$. The complexity of *IFT* is the same as that of the *FFT*.

8.5.3.4 Computing Product of Polynomials

There is an efficient way to compute products of polynomials using *FFT* and *IFT*. Write $p(x)$ and $q(x)$ for the values of polynomial p and q at point x , and $p \times q$ for the product of p and q . Then $(p \times q)(x) = p(x) \times q(x)$, that is, the value of the product polynomial at x is the product of the values of the component polynomials at x . Therefore, compute the values of p and q at the primitive roots of 1, that is $(FFT\ p)$ and $(FFT\ q)$. Then take their products at each point, $(FFT\ p) \times (FFT\ q)$, which is $(FFT\ (p \times q))$. Next, apply *IFT* to this value to get $(p \times q)$. This process is summarized by the equation:

$$(p \times q) = IFT(FFT\ (p \times q)) = IFT((FFT\ p) \times (FFT\ q)).$$

This computation takes only $\mathcal{O}(N \log N)$ sequential time in contrast to $\mathcal{O}(N^2)$ that would be needed for a straightforward multiplication of polynomials of degree N .

Multiplying two numbers can be viewed as multiplication of polynomials. So, this technique can be used to get an $\mathcal{O}(N \log N)$ sequential algorithm for multiplication of N -digit numbers [see Schönage and Strassen 1971].

8.5.4 Batchier Sorting Schemes

Batchier [1968] has devised a sorting method that uses merging, as in merge sort, but better suited for parallel programming. The compactness of the description of Batchier's sorting schemes demonstrates the importance of treating recursion and parallelism simultaneously.

Henceforth, a list is *sorted* means that its elements are arranged in non-decreasing (ascending) order. The algorithm uses an infix compare-and-exchange operator, $\uparrow$, that is applied to a pair of equal length powerlists, p and q . It creates a single powerlist $p \uparrow q$ where the $2i^{\text{th}}$ and $(2i + 1)^{\text{th}}$ items of $p \uparrow q$ are $(p_i \min q_i)$ and $(p_i \max q_i)$, respectively.

$$p \uparrow q = (p \min q) \bowtie (p \max q)$$

where $\min$ and $\max$ are infix comparison operators over scalar values with their standard meanings. Thus, $\langle 5\ 2 \rangle \uparrow \langle 4\ 3 \rangle = \langle 4\ 5\ 2\ 3 \rangle$.

Using $\mathcal{O}(|p|)$ processors, $p \uparrow q$ can be computed in constant parallel time, and in $\mathcal{O}(|p|)$ sequential time using a single processor.

8.5.4.1 Batchers Odd-Even Mergesort

Below, *boesort*, for *batcher-odd-even-sort*, sorts a powerlist using the helper function *bm*, written as a binary infix operator, that merges two sorted powerlists in parallel. The code of *boesort* is unremarkable, that of *bm* remarkable.

$$\begin{aligned} \text{boesort}\langle x \rangle &= \langle x \rangle \\ \text{boesort}(p \bowtie q) &= (\text{boesort } p) \text{bm} (\text{boesort } q) \end{aligned}$$

$$\begin{aligned} \langle x \rangle \text{bm} \langle y \rangle &= \langle x \rangle \uparrow \langle y \rangle \\ (r \bowtie s) \text{bm} (u \bowtie v) &= (r \text{bm} v) \uparrow (s \text{bm} u) \end{aligned}$$

I prove the correctness of *boesort* in Section 8.5.4.3 (page 482).

Complexity The sequential complexity of *boesort* can be analyzed as follows. Let $Bm(n)$ be the sequential complexity of applying *bm* to two lists of size n . Taking the sequential complexity of $\uparrow$ to be $\mathcal{O}(n)$ for n items:

$$\begin{aligned} Bm(1) &= 1 \\ Bm(n) &= 2.Bi(n/2) + \mathcal{O}(n) \end{aligned}$$

From the Master theorem (see Section 4.10.2.3, page 186), $Bm(n) = \mathcal{O}(n \log n)$. A similar recurrence for *boesort* is:

$$\begin{aligned} \text{Boesort}(1) &= 1 \\ \text{Boesort}(n) &= 2.(\text{Boesort}(n/2) + Bm(n)) \end{aligned}$$

Solving this recurrence by expansion (see Section 4.10.2.2, page 185), $\text{Boesort}(n) = \mathcal{O}(n \log^2 n)$. The parallel complexity, assuming $\mathcal{O}(n)$ processors are available, is $Bm(n) = Bm(n/2) + \mathcal{O}(1)$, that is, $\log n$. And, $\text{Boesort}(n) = \text{Boesort}(n/2) + Bm(n)$, that is, $\mathcal{O}(\log^2 n)$. So, *boesort* can be implemented efficiently in parallel.

8.5.4.2 Elementary Facts About Sorting

A “compare-and-exchange” type sorting method merely exchanges the elements but never alters their values. For analysis of such methods, it is often useful to employ the zero-one principle, described next.

Zero-one principle Define a *sorting schedule* for array A to be a sequence of pairs $\langle (i_0, j_0), (i_1, j_1), \dots (i_t, j_t), \dots (i_m, j_m) \rangle$ where i_t and j_t , for all t , are indices in A . Each pair (i_t, j_t) denotes the command

if $A[i_t] > A[j_t]$ **then** $A[i_t], A[j_t] := A[j_t], A[i_t]$ **else skip endif**

The schedule is correct if it sorts A in ascending order for all possible values of its elements. Note that a schedule only permutes the elements of A but does not alter any value.

The zero-one principle says that the schedule sorts A for all possible values of its elements iff it sorts A whenever the values are either 0 or 1.

Proof of zero-one principle Assume that a schedule sorts 0-1 valued arrays. If A consists of arbitrary values, we show that for any x and y in A , $x < y$, the index of x will be smaller than y after application of the schedule; so, the schedule sorts A for arbitrary values.

Construct array B in which element $B[i]$ is a pair $(0, A[i])$ if $A[i] \leq x$ and $(1, A[i])$ if $A[i] > x$. Apply the sorting schedule replacing each reference to A by B and treating $>$ as the comparison over the first component of a pair. The schedule mimics exchanges in A with 0-1 values; so it sorts B on the first components. Since x and y have associated first components 0 and 1, respectively, x will precede y in the array. ■

Function *boesort* applies only the compare-and-swap operation $\uparrow$; so the zero-one principle is applicable in its analysis.² Henceforth, I consider only 0-1 powerlists and show that *boesort* sorts all such powerlists.

Properties of 0-1 powerlists Let z be a function that returns the number of 0's in a 0-1 powerlist. To simplify notation, write zp and zq for $(z\ p)$ and $(z\ q)$. I enumerate a number of properties of 0-1 powerlists.

1. Counting the number of zeroes:

- (a) $z\langle x \rangle$ is either 0 or 1.
- (b) $z(p \mid q) = zp + zq$, $z(p \bowtie q) = zp + zq$
- (c) $z(p \uparrow q) = zp + zq$
- (d) $z(p \text{ } bm \text{ } q) = zp + zq$

2. Actually, being a function *boesort* does not alter its arguments. But the same idea applies in its analysis.

2. Properties of sorted powerlists:

- (a) $\langle x \rangle$ sorted, $\langle x \rangle \uparrow \langle y \rangle$ sorted,
- (b) $(p \bowtie q)$ sorted $\equiv p$ sorted $\wedge q$ sorted $\wedge 0 \leq zp - zq \leq 1$,
- (c) p sorted, q sorted, $zp \geq zq \Rightarrow (p \min q) = p \wedge (p \max q) = q$,
- (d) p sorted, q sorted, $|zp - zq| \leq 1 \Rightarrow (p \updownarrow q)$ sorted.

Note: The condition analogous to (2b) under which $p \mid q$ is sorted is: p sorted $\wedge q$ sorted $\wedge (zp < (len\ p) \Rightarrow zq = 0)$. The simplicity of (2b) suggests why zip is the primary operator in parallel sorting.

Proofs of properties of 0–1 powerlists The assertions in (1) and (2a) are simple to prove. To prove the other assertions in (2), observe that a sorted 0–1 powerlist is of the form $a ++ b$, where segment a is all 0's and b is all 1's; either segment may be empty. Given that $(p \bowtie q) = a ++ b$ which is sorted, the elements of p and q alternate along $a ++ b$ as shown below; here I write p for each occurrence of an element

of p and similarly q : $\overbrace{pqppqpqpq}^{0's \quad 1's}$.

Proof of (2b) From above, both p and q are sorted. For the number of zeroes, note that if the length of a is even, $zp = zq$, otherwise $zp = zq + 1$. Proof of (2c) is similar.

Proof of (2d) Since the statement of (2d) is symmetric in p and q , assume $zp \geq zq$.

$$\begin{aligned}
 & p \text{ sorted, } q \text{ sorted, } |zp - zq| \leq 1 \\
 \Rightarrow & \{ \text{assumption: } zp \geq zq \} \\
 & p \text{ sorted, } q \text{ sorted, } 0 \leq zp - zq \leq 1 \\
 \Rightarrow & \{ \text{from (2b) and (2c)} \} \\
 & p \bowtie q \text{ sorted, } (p \min q) = p, (p \max q) = q \\
 \Rightarrow & \{ \text{replace } p \text{ and } q \text{ in } p \bowtie q \text{ above by } (p \min q) \text{ and } (p \max q) \} \\
 & (p \min q) \bowtie (p \max q) \text{ sorted} \\
 \Rightarrow & \{ \text{definition of } p \updownarrow q \} \\
 & p \updownarrow q \text{ sorted}
 \end{aligned}$$

8.5.4.3 Correctness of Batcher Odd-Even Mergesort

I first prove that bm merges two sorted powerlists to form a sorted powerlist.

Theorem 8.1 If p and q are sorted, $p\ bm\ q$ is sorted.

Proof. Proof is by structural induction.

- Base case, $p = \langle x \rangle$, $q = \langle y \rangle$: $\langle x \rangle\ bm\ \langle y \rangle = \langle x \rangle \updownarrow \langle y \rangle$. From (2a), $\langle x \rangle \updownarrow \langle y \rangle$ is sorted.
- Inductive case, $p = (r \bowtie s)$, $q = (u \bowtie v)$:

$$\begin{aligned}
& (r \bowtie s) \text{ sorted}, (u \bowtie v) \text{ sorted} \\
\equiv & \{ \text{from (2b)} \} \\
& r \text{ sorted}, s \text{ sorted}, 0 \leq zr - zs \leq 1, \\
& u \text{ sorted}, v \text{ sorted}, 0 \leq zu - zv \leq 1 \\
\Rightarrow & \{(zr + zv) - (zs + zu) = (zr - zs) + (zv - zu)\}. \\
& 0 \leq zr - zs \leq 1 \text{ and } -1 \leq zv - zu \leq 0 \text{ implies} \\
& -1 \leq (zr - zs) + (zv - zu) \leq 1 \\
& r \text{ sorted}, s \text{ sorted}, u \text{ sorted}, v \text{ sorted}, |(zr + zv) - (zs + zu)| \leq 1 \\
\Rightarrow & \{ \text{induction: } (r \text{ } bm \text{ } v) \text{ sorted}, (s \text{ } bm \text{ } u) \text{ sorted.} \\
& \text{From (1d)} \ z(r \text{ } bm \text{ } v) = zr + zv, z(s \text{ } bm \text{ } u) = zs + zu \} \\
& (r \text{ } bm \text{ } v) \text{ sorted}, (s \text{ } bm \text{ } u) \text{ sorted}, |z(r \text{ } bm \text{ } v) - z(s \text{ } bm \text{ } u)| \leq 1 \\
\Rightarrow & \{ \text{from the definition of } bm \} \\
& (r \bowtie s) \text{ } bm \text{ } (u \bowtie v) \text{ sorted} \\
\Rightarrow & \{ \text{from (2d)} \} \\
& (r \text{ } bm \text{ } v) \uparrow (s \text{ } bm \text{ } u) \text{ sorted}
\end{aligned}$$

Theorem 8.2 For any powerlist p , $boesort\ p$ is sorted.

Proof. Proof is by structural induction. For $p = \langle x \rangle$, $boesort\ \langle x \rangle = \langle x \rangle$, and $\langle x \rangle$ is sorted. For $p = u \bowtie v$

$$\begin{aligned}
& p = u \bowtie v \\
\Rightarrow & \{ \text{definition of } boesort \} \\
& boesort\ p = (boesort\ u) \text{ } bm \text{ } (boesort\ v) \\
\Rightarrow & \{ \text{inductively, } (boesort\ u) \text{ and } (boesort\ v) \text{ are sorted.} \\
& \text{From Theorem 8.1 } (boesort\ u) \text{ } bm \text{ } (boesort\ v) \text{ is sorted.} \} \\
& (boesort\ p) \text{ sorted}
\end{aligned}$$

8.5.4.4 Bitonic Sort

A sequence of the form $Y ++ Z$ where Y is a sorted sequence in ascending order and Z in descending order, or Y descending and Z ascending, is called *bitonic*. Either Y or Z could be empty; so, a sorted sequence is bitonic. We can form a bitonic sequence from two sorted linear sequences U and V by forming $U ++ rev(V)$. Below, $bisort$ sorts a powerlist using the helper function bi that sorts a bitonic powerlist. As before, $bisort$ is unremarkable and bi is remarkable.

$$\begin{aligned}
bisort\langle x \rangle &= \langle x \rangle \\
bisort(p \mid q) &= bi((bisort\ p) \mid revp(bisort\ q)) \\
bi\langle x \rangle &= \langle x \rangle \\
bi(p \bowtie q) &= (bi\ p) \updownarrow (bi\ q)
\end{aligned}$$

Complexity The sequential complexity of *bi* is given by the equations:

$$Bi(1) = 1, \quad Bi(n) = 2 \cdot Bi(n/2) + \mathcal{O}(n).$$

So, $Bi(n) = \mathcal{O}(n \log n)$. A similar recurrence for *bisort* is:

$$\begin{aligned}
Bisort(1) &= 1 \\
Bisort(n) &= 2 \cdot (Bisort(n/2) + \mathcal{O}(n)) + Bi(n)
\end{aligned}$$

where $\mathcal{O}(n)$ in the first term is for the sequential implementation of *revp*. Solving this recurrence by expansion, see Section 4.10.2.2 (page 185), $Bisort(n) = \mathcal{O}(n \log^2 n)$. The parallel complexity, assuming $\mathcal{O}(n)$ processors are available, is $Bi(n) = Bi(n/2) + \mathcal{O}(1)$, that is, $\log n$. And, $Bisort(n) = (Bisort(n/2) + \mathcal{O}(1)) + Bi(n)$, that is, $\mathcal{O}(\log^2 n)$. So, *bisort* can be implemented efficiently in parallel.

Correctness of bitonic sort We first establish a relationship between functions *bi* and *bm* for arbitrary powerlists, ones that are neither sorted nor bitonic. The correctness of *bisort* follows quite easily then.

Theorem 8.3 For arbitrary powerlists *p* and *q*,

$$bi(p \mid revp\ q) = p\ bm\ q.$$

Proof. Proof is by structural induction.

- Base case, $bi(\langle x \rangle \mid revp\langle y \rangle) = \langle x \rangle\ bm\ \langle y \rangle$:

$$\begin{aligned}
&bi(\langle x \rangle \mid revp\langle y \rangle) \\
&= \{\text{definition of } revp\} \\
&\quad bi(\langle x \rangle \mid \langle y \rangle) \\
&= \{(\langle x \rangle \mid \langle y \rangle) = (\langle x \rangle \bowtie \langle y \rangle)\} \\
&\quad bi(\langle x \rangle \bowtie \langle y \rangle) \\
&= \{\text{definition of } bi\} \\
&\quad \langle x \rangle \updownarrow \langle y \rangle \\
&= \{\text{definition of } bm\} \\
&\quad \langle x \rangle\ bm\ \langle y \rangle
\end{aligned}$$

- Inductive case, $bi(p \mid (revp\ q)) = p\ bm\ q$: Let $p = r \bowtie s$ and $q = u \bowtie v$.

$$\begin{aligned}
& bi(p \mid (revp\ q)) \\
= & \{\text{expanding } p, q\} \\
& bi((r \bowtie s) \mid revp(u \bowtie v)) \\
= & \{\text{definition of } revp\} \\
& bi((r \bowtie s) \mid ((revp\ v) \bowtie (revp\ u))) \\
= & \{\mid, \bowtie \text{ distribute}\} \\
& bi((r \mid (revp\ v)) \bowtie (s \mid (revp\ u))) \\
= & \{\text{definition of } bi\} \\
& bi(r \mid (revp\ v)) \updownarrow bi(s \mid (revp\ u)) \\
= & \{\text{induction}\} \\
& (r\ bm\ v) \updownarrow (s\ bm\ u) \\
= & \{\text{definition of } bm\} \\
& (r \bowtie s)\ bm\ (u \bowtie v) \\
= & \{\text{using the definitions of } p, q\} \\
& p\ bm\ q
\end{aligned}$$

Theorem 8.4 For any powerlist p , $(bisort\ p)$ is sorted.

Proof. Proof is by structural induction. From its definition, $bisort\langle x \rangle = \langle x \rangle$, and $\langle x \rangle$ is sorted. For the inductive case,

$$\begin{aligned}
& bisort(p \mid q) \\
= & \{\text{definition of } bisort\} \\
& bi((bisort\ p) \mid revp(bisort\ q)) \\
= & \{\text{from Theorem 8.3}\} \\
& (bisort\ p)\ bm\ (bisort\ q) \\
= & \{\text{inductively } (bisort\ p) \text{ and } (bisort\ q) \text{ are sorted}\} \\
& u\ bm\ v, \text{ where } u \text{ and } v \text{ are sorted.}
\end{aligned}$$

From Theorem 8.1, $(u\ bm\ v)$, and hence $bisort(p \mid q)$, is sorted. ■

8.5.5 Prefix Sum

I introduced the prefix sum problem in Section 7.4.3.3 (page 419) over linear lists. Here I develop parallel algorithms for the problem on powerlists. I show one application, in designing a carry-lookahead adder circuit in Section 8.5.5.7 (page 490).

Start with a domain D over which $\oplus$ is an associative binary operator; $\oplus$ need not be commutative. Let L be a powerlist $\langle x_0, x_1, \dots, x_i, \dots, x_N \rangle$ over D . The prefix sum,

$psum$, of L with respect to $\oplus$ is:

$$psum \langle x_0, x_1, \dots, x_i, \dots, x_N \rangle = \langle x_0, x_0 \oplus x_1, \dots, x_0 \oplus x_1 \oplus \dots \oplus x_i, \dots, x_0 \oplus x_1 \oplus \dots \oplus x_N \rangle.$$

That is, element i of $(psum L)$ is obtained by applying $\oplus$ in order to all elements of L with index smaller than or equal to i . The length of $(psum L)$ is the same as that of L . Being a scalar function, $\oplus$ distributes over both powerlist constructors, $|$ and $\bowtie$.

8.5.5.1 Specification

Right-shift operator I postulate that $\oplus$ has a left identity element $\mathbf{0}$ in D ; so, $\mathbf{0} \oplus x = x$, for all x in D . The introduction of $\mathbf{0}$ is not strictly necessary.³ For powerlist p , $p^\rightarrow$ is the powerlist obtained by shifting p to the right by one position. The effect of shifting is to append $\mathbf{0}$ as the first element and discard the rightmost element of p ; thus, $\langle a b c d \rangle^\rightarrow = \langle \mathbf{0} a b c \rangle$. Formally,

$$\begin{aligned} \langle x \rangle^\rightarrow &= \langle \mathbf{0} \rangle \\ (p \bowtie q)^\rightarrow &= q^\rightarrow \bowtie p \end{aligned}$$

Henceforth, $\rightarrow$ has higher binding power than other function applications and powerlist constructors. So, $p^\rightarrow \oplus q^\rightarrow \bowtie v^\rightarrow$ means $((p^\rightarrow) \oplus (q^\rightarrow)) \bowtie (v^\rightarrow)$.

It is easy to prove that $(p \oplus q)^\rightarrow = p^\rightarrow \oplus q^\rightarrow$. The following lemma is useful in developing an algorithm for the prefix sum problem.

Lemma 8.1 Let $L = p \bowtie q$. Then $L^\rightarrow \oplus L = (q^\rightarrow \oplus p) \bowtie (p \oplus q)$.

Proof.

$$\begin{aligned} &L^\rightarrow \oplus L \\ = &\{L = p \bowtie q\} \\ &(p \bowtie q)^\rightarrow \oplus (p \bowtie q) \\ = &\{(p \bowtie q)^\rightarrow = (q^\rightarrow \bowtie p)\} \\ &(q^\rightarrow \bowtie p) \oplus (p \bowtie q) \\ = &\{\oplus, \bowtie \text{ distribute}\} \\ &(q^\rightarrow \oplus p) \bowtie (p \oplus q) \end{aligned}$$

■

Specification of prefix sum Let L_i be the i^{th} element of L , $0 \leq i < \text{length}(L)$. From the informal description of $psum$:

$$\begin{aligned} (psum L)_0 &= L_0 \\ (psum L)_i &= (psum L)_{i-1} \oplus L_i, \text{ for } i > 0, \text{ that is,} \\ (psum L)_i &= (psum L)_i^\rightarrow \oplus L_i, \text{ for } i > 0 \end{aligned}$$

3. Adams [1994] has specified the prefix-sum problem without postulating an explicit $\mathbf{0}$ element. He introduces a binary operator $\tilde{\oplus}$ over two similar powerlists such that $p \tilde{\oplus} q = p^\rightarrow \oplus q$. He defines $\tilde{\oplus}$ without introducing $\mathbf{0}$.

The last equation holds for $i = 0$ as well because $(psum L)_0^\rightarrow \oplus L_0 = \mathbf{0} \oplus L_0 = L_0 = (psum L)_0$. The formal specification of $psum$ is derived from these observations:

$$(psum L) = (psum L)^\rightarrow \oplus L \quad (\text{E.psum})$$

Equation (E.psum) is illustrated in Table 8.1 for $L = \langle a \ b \ c \ d \rangle$.

Table 8.1 Equation (E.psum) illustrated

L	$\langle a$	b	c	d	$\rangle$
$(psum L)$	$\langle a$	$a \oplus b$	$a \oplus b \oplus c$	$a \oplus b \oplus c \oplus d$	$\rangle$
$(psum L)^\rightarrow$	$\langle \mathbf{0}$	a	$a \oplus b$	$a \oplus b \oplus c$	$\rangle$
$(psum L)^\rightarrow \oplus L$	$\langle a$	$a \oplus b$	$a \oplus b \oplus c$	$a \oplus b \oplus c \oplus d$	$\rangle$

8.5.5.2 Properties of Prefix Sum

Lemma 8.3 $psum \langle x \rangle = \langle x \rangle$

Proof.

$$\begin{aligned}
 & psum \langle x \rangle \\
 = & \{\text{substitute } \langle x \rangle \text{ for } L \text{ in (E.psum)}\} \\
 & (psum \langle x \rangle)^\rightarrow \oplus \langle x \rangle \\
 = & \{(psum \langle x \rangle) \text{ is a singleton. So, } (psum \langle x \rangle)^\rightarrow = \langle \mathbf{0} \rangle\} \\
 & \langle \mathbf{0} \rangle \oplus \langle x \rangle \\
 = & \{\mathbf{0} \text{ is a zero of } \oplus\} \\
 & \langle x \rangle
 \end{aligned}$$

Theorem 8.5 Let $psum(p \bowtie q) = r \bowtie t$. Then

$$r = t^\rightarrow \oplus p = psum(q^\rightarrow \oplus p) \quad (\text{psum1})$$

$$t = r \oplus q = psum(p \oplus q) \quad (\text{psum2})$$

Proof.

$$\begin{aligned}
 & psum(p \bowtie q) \\
 = & \{\text{substitute } (p \bowtie q) \text{ for } L \text{ in (E.psum)}\} \\
 & psum(p \bowtie q)^\rightarrow \oplus (p \bowtie q) \\
 = & \{psum(p \bowtie q) = r \bowtie t\} \\
 & (r \bowtie t)^\rightarrow \oplus (p \bowtie q) \\
 = & \{(r \bowtie t)^\rightarrow = t^\rightarrow \bowtie r\} \\
 & (t^\rightarrow \bowtie r) \oplus (p \bowtie q) \\
 = & \{\oplus, \bowtie \text{ distribute}\} \\
 & (t^\rightarrow \oplus p) \bowtie (r \oplus q)
 \end{aligned}$$

Therefore, $psum(p \bowtie q) = r \bowtie t = (t^\rightarrow \oplus p) \bowtie (r \oplus q)$. Applying law L2 (unique deconstruction), conclude $r = t^\rightarrow \oplus p$, and $t = r \oplus q$, which prove the first parts

of the identities in (psum1) and (psum2). Next, I solve these two equations in unknowns r and t to establish the remaining identities.

$$\begin{aligned}
& r = t^{\rightarrow} \oplus p, \quad t = r \oplus q \\
\Rightarrow & \{\text{substitute for } t \text{ in the equation for } r, \text{ and for } r \text{ in the equation for } t\} \\
& r = (r \oplus q)^{\rightarrow} \oplus p, \quad t = t^{\rightarrow} \oplus (p \oplus q) \\
\Rightarrow & \{(r \oplus q)^{\rightarrow} = r^{\rightarrow} \oplus q^{\rightarrow}, \text{ and } \oplus \text{ is associative}\} \\
& r = r^{\rightarrow} \oplus (q^{\rightarrow} \oplus p), \quad t = t^{\rightarrow} \oplus (p \oplus q) \\
\Rightarrow & \{\text{Apply (E.psum) on the equations for } r \text{ and } t\} \\
& r = \text{psum}(q^{\rightarrow} \oplus p), \quad t = \text{psum}(p \oplus q)
\end{aligned}$$

Corollary 8.1 For non-singleton powerlist L :

$$\text{psum } L = (\text{psum } u) \bowtie (\text{psum } v), \text{ where } u \bowtie v = L^{\rightarrow} \oplus L.$$

Proof. Let $L = p \bowtie q$.

$$\begin{aligned}
& \text{psum } L \\
= & \{L = p \bowtie q\} \\
& \text{psum}(p \bowtie q) \\
= & \{\text{from Theorem 8.5}\} \\
& \text{psum}(q^{\rightarrow} \oplus p) \bowtie \text{psum}(p \oplus q) \\
= & \{\text{Given } u \bowtie v = L^{\rightarrow} \oplus L. \\
& \text{From Lemma 8.1 (page 486), } L^{\rightarrow} \oplus L = (q^{\rightarrow} \oplus p) \bowtie (p \oplus q). \\
& \text{So, } u = (q^{\rightarrow} \oplus p) \text{ and } v = (p \oplus q).\} \\
& (\text{psum } u) \bowtie (\text{psum } v)
\end{aligned}$$

The next corollary follows directly from Theorem 8.5. ■

Corollary 8.2 $\text{psum}(p \bowtie q) = (t^{\rightarrow} \oplus p) \bowtie t$, where $t = \text{psum}(p \oplus q)$.

8.5.5.3 A Simple Algorithm for Prefix Sum

A particularly simple scheme to compute prefix sum of L of 2^n elements is implemented on $\mathcal{O}(2^n)$ processors as follows. Initially, processor i has L_i as its data item, $0 \leq i < 2^n$. In step j , $0 \leq j < n$, it sends its data to processor $i + 2^j$, provided such a processor exists; so processor k receives no data in step j if $k < 2^j$. On receiving data r in a step, a processor updates its own data d to $r \oplus d$; if it receives no data then d is unchanged. The computation terminates after $n - 1$ steps. Then the data value at processor i is $(\text{psum } L)_i$. Function *sps* (simple prefix sum) implements this scheme.

$$\begin{aligned}
sps \langle x \rangle &= \langle x \rangle \\
sps L &= (sps u) \bowtie (sps v) \\
\text{where } u \bowtie v &= L^{\rightarrow} \oplus L
\end{aligned}$$

Correctness of sps is a direct consequence of Corollary 8.1 (page 488).

8.5.5.4 Ladner–Fischer Algorithm

A scheme, due to [Ladner and Fischer \[1980\]](#), first applies $\oplus$ to adjacent elements x_{2i}, x_{2i+1} to compute the list $\langle x_0 \oplus x_1, \dots, x_{2i} \oplus x_{2i+1}, \dots \rangle$. This list has half as many elements as the original list; its prefix sum is then computed recursively. The resulting list is $\langle x_0 \oplus x_1, \dots, x_0 \oplus x_1 \oplus \dots \oplus x_{2i} \oplus x_{2i+1}, \dots \rangle$. This list contains half of the elements of the final list; the missing elements are $x_0, x_0 \oplus x_1 \oplus x_2, \dots, x_0 \oplus x_1 \oplus \dots \oplus x_{2i}, \dots$. These elements can be computed by “adding” $x_2, x_4, \dots$, appropriately to the elements of the already computed list. The Ladner–Fischer scheme is defined by function lf .

$$\begin{aligned}
lf \langle x \rangle &= \langle x \rangle \\
lf(p \bowtie q) &= (t^{\rightarrow} \oplus p) \bowtie t \\
\text{where } t &= lf(p \oplus q)
\end{aligned}$$

Correctness of lf is a direct consequence of Corollary 8.2 (page 488).

8.5.5.5 Complexity of the Ladner–Fischer Algorithm

The simple scheme, sps , is not as efficient as the Ladner–Fischer algorithm, lf , because there are two recursive calls in sps whereas lf has just one. For the sequential complexity of lf , solve the recurrence $S(1) = \Theta(1)$, $S(n) = S(n/2) + \Theta(n)$ to get $S(n) = \Theta(n)$, using the Master theorem (Section 4.10.2.3, page 186).

For parallel complexity, let $P(n)$ be the number of steps to compute lf on a powerlist of length n . Then $P(1) = \Theta(1)$. And, $P(n) = P(n/2) + \Theta(\log n)$ because computation of t takes $P(n/2)$ steps and $t^{\rightarrow}$ a further $\Theta(\log n)$. So, $P(n) = \Theta(\log^2 n)$. However, we can actually compute lf in $\Theta(\log n)$ parallel time, by computing $(lf u)$ and $(lf u)^{\rightarrow}$ together, for any powerlist u , as shown below.

Let $u = p \bowtie q$. We use Theorem 8.5 (page 487) and its corollaries. From Corollary 8.2 (page 488), $lf(p \bowtie q) = (t^{\rightarrow} \oplus p) \bowtie t$, where $t = lf(p \oplus q)$. And $(lf(p \bowtie q))^{\rightarrow} = (r \bowtie t)^{\rightarrow} = t^{\rightarrow} \bowtie r$ where $r = t^{\rightarrow} \oplus p$. So, $(lf(p \bowtie q))^{\rightarrow} = t^{\rightarrow} \bowtie (t^{\rightarrow} \oplus p)$. Let $Q(n)$ be the number of steps to compute $lf(p \bowtie q)$ and $(lf(p \bowtie q))^{\rightarrow}$ simultaneously. Both computations need t and $t^{\rightarrow}$; so, $Q(n) = Q(n/2) + \Theta(1)$, or $Q(n) = \Theta(\log n)$.

Additionally, lf can be implemented in parallel using only $n/\log n$ processors. So, the total work, given by the number of processors times the computation time, is $n/\log n \times \Theta(\log n) = \Theta(n)$, the same as the sequential complexity. Parallel algorithms with this property are called *work-efficient*.

8.5.5.6 Remarks on Prefix Sum Description

A more traditional way of describing a prefix sum algorithm, such as the simple scheme *sps*, is to explicitly name the quantities that are being computed and establish relationships among them. Let y_{ij} be computed by the i^{th} processor at the j^{th} step. Then, for all i and j , $0 \leq i, j < 2^n$,

$$y_{i0} = x_i, \text{ and}$$

$$y_{ij+1} = \left\{ \begin{array}{ll} y_{i-2^j j} & , \quad i \geq 2^j \\ 0 & , \quad i < 2^j \end{array} \right\} \oplus y_{ij}$$

The correctness criterion is $y_{i,n} = x_0 \oplus \cdots \oplus x_i$, for all i . This specification is considerably more difficult to prove by algebraic manipulation. The Ladner–Fischer scheme is even more difficult to specify in this manner. Algebraic methods are simpler for describing uniform operations on aggregates of data.

8.5.5.7 Carry-Lookahead adder

I show the application of prefix sum in designing a carry-lookahead adder. An adder circuit in a computer adds two numbers, invariably represented by binary strings of equal length. Henceforth, represent each number by a powerlist of bits where the bits are in order from the lowest to highest, reverse of the usual representation, so that the prefix sum algorithm agrees with the circuit description. An example is shown in Table 8.2, where the powers of 2 and the representations of numbers $p = 198$ and $q = 92$ are shown in the top three rows.

Table 8.2 Examples of relevant powerlists in a carry-lookahead adder

2^n	<	1	2	4	8	16	32	64	128	>
$p = 198$	<	0	1	1	0	0	0	1	1	>
$q = 92$	<	0	0	1	1	1	0	1	0	>
d	<	0	π	1	π	π	0	1	π	>
c	<	0	0	0	1	1	1	0	1	>
$s = 34$	<	0	1	0	0	0	1	0	0	> 1

Mechanism of addition The main part of the algorithm is the computation of the carries, represented by powerlist c , where c_i is the carry into position i . Then the i^{th} bit of the sum, s_i , is $c_i +_2 p_i +_2 q_i$, where $+_2$ is addition modulo 2. There may be a carry out of the last position, an overflow bit, which I ignore in the following discussion; the overflow can be prevented if the last two bits in p and q are both 0's. The powerlists c and s are shown as the last two rows in Table 8.2 where the overflow bit out of s is shown outside powerlist s . Observe that s can be computed in $\mathcal{O}(1)$ parallel steps after c has been computed.

The remaining part of this section is devoted to efficient computation of the carries, c . Clearly, $c_0 = 0$. And c_{i+1} is the carry from $c_i +_2 p_i +_2 q_i$. It follows that if p_i and q_i are (1) both 1's then $c_{i+1} = 1$ independent of the value of c_i , (2) both 0's then $c_{i+1} = 0$ independent of the value of c_i , and (3) 0 and 1 in either order then $c_{i+1} = c_i$. Therefore, a pair of bits (p_i, q_i) are called *carry generating*, *carry absorbing* or *carry propagating* if their values are (1, 1), (0, 0), and (0, 1) or (1, 0), respectively.

To facilitate the description, define infix binary operator $\otimes$ over bit values that returns 1, 0 or π to denote carry generating, carry absorbing or carry propagating pairs of bits, respectively.

$$x \otimes y = \begin{cases} x & \text{if } x = y, \\ \pi & \text{otherwise} \end{cases}$$

Then encode each pair of input bits (p_i, q_i) by $d_i = p_i \otimes q_i$, so $d = p \otimes q$. Computation of d takes $\mathcal{O}(1)$ parallel steps. In Table 8.2, value of d is shown in the row above c .

Carry computation as prefix sum Define infix binary operator $\oplus$ over the values $\{0, 1, \pi\}$:

$$x \oplus y = \begin{cases} x & \text{if } y = \pi, \\ y & \text{otherwise} \end{cases}$$

The zero of $\oplus$, 0 , is just 0. Note that $\oplus$ is not commutative. From the earlier discussion about the mechanism of addition, the carry powerlist c is:

$$c_0 = 0 \text{ and } c_{i+1} = c_i \oplus d_i. \quad (\text{Adder0})$$

Lemma 8.3 $c = (\text{psum } d^\rightarrow)$ with respect to $\oplus$, that is, $c_i = (\text{psum } d^\rightarrow)_i$ for all i .

Proof. First, I prove a simple result about $(\text{psum } d^\rightarrow)$.

$$\begin{aligned} & (\text{psum } d^\rightarrow) \\ &= \{\text{Apply equation (E.psum) to } (\text{psum } d^\rightarrow)\} \\ & \quad (\text{psum } d^\rightarrow)^\rightarrow \oplus d^\rightarrow \\ &= \{\text{property of } \rightarrow \text{ and scalar } \oplus\} \\ & \quad ((\text{psum } d^\rightarrow) \oplus d)^\rightarrow \end{aligned}$$

$$\text{Therefore, } (\text{psum } d^\rightarrow)_i = ((\text{psum } d^\rightarrow) \oplus d)_i^\rightarrow, \text{ for all } i \quad (\text{Adder1})$$

Next, I show $c_i = (\text{psum } d^\rightarrow)_i$, for all i , by induction on i .

• Base case, $c_0 = (\text{psum } d^\rightarrow)_0$:

Set $i = 0$ in (Adder1) to conclude $(\text{psum } d^\rightarrow)_0 = ((\text{psum } d^\rightarrow) \oplus d)_0^\rightarrow$. The right side is 0 because $u_0^\rightarrow = 0$ for any u . So, $(\text{psum } d^\rightarrow)_0 = 0 = c_0$, from (Adder0).

- Inductive case, $c_{i+1} = (psum\ d^{\rightarrow})_{i+1}$:

$$\begin{aligned}
& (psum\ d^{\rightarrow})_{i+1} \\
= & \{\text{from (Adder1)}\} \\
& ((psum\ d^{\rightarrow}) \oplus d)_{i+1}^{\rightarrow} \\
= & \{\text{property of the shift operator } \rightarrow\} \\
& ((psum\ d^{\rightarrow}) \oplus d)_i \\
= & \{\text{rewrite}\} \\
& (psum\ d^{\rightarrow})_i \oplus d_i \\
= & \{\text{inductive hypothesis: } c_i = (psum\ d^{\rightarrow})_i\} \\
& c_i \oplus d_i \\
= & \{\text{from (Adder0)}\} \\
& c_{i+1}
\end{aligned}$$

The adder algorithm Putting the pieces together, the adder algorithm is:

$$\begin{aligned}
adder\ p\ q &= c \oplus_2 p \oplus_2 q \\
\text{where } c &= (psum\ d^{\rightarrow}) \\
d &= p \otimes q
\end{aligned}$$

■

8.6 Multi-dimensional Powerlist

A major part of parallel computing involves arrays of one or more dimensions. For an array of one dimension, a simple, unnested powerlist is sufficient, and its length is the only measure that is required for proofs by structural induction. A multi-dimensional array is encoded by a multi-dimensional powerlist (*mdp*). Such a powerlist has a finite number of dimensions and possibly different lengths in different dimensions. Proofs about these structures require induction at several levels.

8.6.1 Multi-dimensional Constructors

An array of m dimensions (dimensions are numbered 0 through $m - 1$) is represented by a multi-dimensional powerlist (*mdp*), a powerlist with $(m - 1)$ levels of nesting. Conversely, any powerlist with $(m - 1)$ levels of nesting may be regarded as an array of dimension m . For example, consider a 2-dimensional matrix of r rows and c columns, where both r and c are powers of 2. It may be stored row-wise so that the powerlist has c elements where each element is a powerlist of length r representing a row. Conversely, the same matrix may be represented by a powerlist of length r where each element is a powerlist of c elements representing a column. Powerlist constructors are designed to be symmetric in treating rows and columns.

Example 8.1 Matrices A and B are shown in the top part of Figure 8.5 in traditional notation. Below, I show their powerlist representations where the matrices are stored column-wise, that is each element of the outer powerlist is a column, which is also a powerlist.

$$\begin{aligned} A &= \langle \langle 2\ 3 \rangle \langle 4\ 5 \rangle \rangle \\ B &= \langle \langle 0\ 6 \rangle \langle 1\ 7 \rangle \rangle \\ A \mid B &= \langle \langle 2\ 3 \rangle \langle 4\ 5 \rangle \langle 0\ 6 \rangle \langle 1\ 7 \rangle \rangle \\ A \bowtie B &= \langle \langle 2\ 3 \rangle \langle 0\ 6 \rangle \langle 4\ 5 \rangle \langle 1\ 7 \rangle \rangle \end{aligned}$$

In manipulating multi-dimensional powerlists, I prefer to think in terms of array operations rather than operations on nested powerlists. Therefore, I introduce constructors $|^n$ and $\bowtie^n$, tie and zip operators for each dimension n . Constructors $|^0$ and $\bowtie^0$ are $\mid$ and $\bowtie$, respectively. Formal definitions for $n > 0$ are given in Section 8.6.1 (page 492).

For a matrix that is stored column-wise, $|^1$ and $\bowtie^1$ apply the corresponding operations on its rows (see Example 8.2).

Example 8.2 In Figure 8.5, $A \mid^0 B = A \mid B$ is the concatenation of A and B by rows, and $A \mid^1 B$ their concatenation by columns. The applications of $\bowtie^0$ and $\bowtie^1$ are analogous. The results of all four operations on A and B are shown as matrices in Figure 8.5 and as powerlists in Figure 8.6.

$$\begin{array}{cc} A = \begin{array}{c} \wedge \quad \wedge \\ \left\langle \begin{array}{cc} 2 & 4 \\ 3 & 5 \end{array} \right\rangle \\ \vee \quad \vee \end{array} & B = \begin{array}{c} \wedge \quad \wedge \\ \left\langle \begin{array}{cc} 0 & 1 \\ 6 & 7 \end{array} \right\rangle \\ \vee \quad \vee \end{array} \\ \\ A \mid^0 B = \begin{array}{c} \wedge \quad \wedge \quad \wedge \quad \wedge \\ \left\langle \begin{array}{cccc} 2 & 4 & 0 & 1 \\ 3 & 5 & 6 & 7 \end{array} \right\rangle \\ \vee \quad \vee \quad \vee \quad \vee \end{array} & A \bowtie^0 B = \begin{array}{c} \wedge \quad \wedge \quad \wedge \quad \wedge \\ \left\langle \begin{array}{cccc} 2 & 0 & 4 & 1 \\ 3 & 6 & 5 & 7 \end{array} \right\rangle \\ \vee \quad \vee \quad \vee \quad \vee \end{array} \\ \\ A \mid^1 B = \begin{array}{c} \wedge \quad \wedge \\ \left\langle \begin{array}{cc} 2 & 4 \\ 3 & 5 \\ 0 & 1 \\ 6 & 7 \end{array} \right\rangle \\ \vee \quad \vee \end{array} & A \bowtie^1 B = \begin{array}{c} \wedge \quad \wedge \\ \left\langle \begin{array}{cc} 2 & 4 \\ 0 & 1 \\ 3 & 5 \\ 6 & 7 \end{array} \right\rangle \\ \vee \quad \vee \end{array} \end{array}$$

Figure 8.5 Constructors over power-matrices.

$$\begin{aligned} A &= \langle \langle 2\ 3 \rangle \langle 4\ 5 \rangle \rangle \\ B &= \langle \langle 0\ 6 \rangle \langle 1\ 7 \rangle \rangle \\ A \mid^0 B &= \langle \langle 2\ 3 \rangle \langle 4\ 5 \rangle \langle 0\ 6 \rangle \langle 1\ 7 \rangle \rangle \\ A \bowtie^0 B &= \langle \langle 2\ 3 \rangle \langle 0\ 6 \rangle \langle 4\ 5 \rangle \langle 1\ 7 \rangle \rangle \\ A \mid^1 B &= \langle \langle 2\ 3\ 0\ 6 \rangle \langle 4\ 5\ 1\ 7 \rangle \rangle \\ A \bowtie^1 B &= \langle \langle 2\ 0\ 3\ 6 \rangle \langle 4\ 1\ 5\ 7 \rangle \rangle \end{aligned}$$

Figure 8.6 Applying constructors over multiple levels in powerlists.

8.6.1.1 Definitions of Multi-dimensional Constructors

I define $|^n$ and $\bowtie^n$ inductively for all n , for $n \geq 0$. Start with $|$ and $\bowtie$ for $|^0$ and $\bowtie^0$, respectively. Constructors $|^n$ or $\bowtie^n$ can be applied on powerlists p and q only if their depths are at least n .

$$\begin{aligned} p |^0 q &= p | q & p \bowtie^0 q &= p \bowtie q \\ \langle p \rangle |^n \langle q \rangle &= \langle p |^{n-1} q \rangle & (p \bowtie p') |^n (q \bowtie q') &= (p |^n q) \bowtie (p' |^n q') \\ \langle p \rangle \bowtie^n \langle q \rangle &= \langle p \bowtie^{n-1} q \rangle & (p | p') \bowtie^n (q | q') &= (p \bowtie^n q) | (p' \bowtie^n q') \end{aligned}$$

8.6.1.2 Properties of Multi-dimensional Constructors

Analogous to the laws of Section 8.2.3 (page 463), I derive several properties of multi-dimensional constructors. See Appendix 8.A.2 (page 506) for proofs.

Theorem 8.6 HL1. (Dual deconstruction)

For powerlists p and q of depth of at least m , there exist u and v such that $u \bowtie^m v = p |^m q$. Conversely, for powerlists u and v of depth of at least m , there exist p and q such that $u \bowtie^m v = p |^m q$.

HL2. (Unique deconstruction) For $m \geq 0$:

$$\begin{aligned} (p |^m p' = q |^m q') &\equiv (p = p' \wedge q = q') \\ (p \bowtie^m p' = q \bowtie^m q') &\equiv (p = p' \wedge q = q') \end{aligned}$$

HL3. (Distributivity of the constructors) For natural numbers m and k , where $m \neq k$:

$$\begin{aligned} (p |^m p') \bowtie^k (q |^m q') &= (p \bowtie^k q) |^m (p' \bowtie^k q') \\ (p |^m p') |^k (q |^m q') &= (p |^k q) |^m (p' |^k q') \\ (p \bowtie^m p') \bowtie^k (q \bowtie^m q') &= (p \bowtie^k q) \bowtie^m (p' \bowtie^k q') \end{aligned}$$

8.6.1.3 Depth, Shape

The number of dimensions of an mdp, the levels of nesting, is given by its *depth*, starting at 0 for a 1-dimensional powerlist.

$$\begin{aligned} \text{depth}\langle s \rangle &= 0 & s \text{ is a scalar} \\ \text{depth}\langle p \rangle &= 1 + (\text{depth } p) & p \text{ is a powerlist} \\ \text{depth}(p | q) &= \text{depth } p \\ \text{depth}(p \bowtie q) &= \text{depth } p \end{aligned}$$

The *shape* of a powerlist is a linear list of pairs, where each pair is the depth and logarithmic length at a particular level.

$$\begin{aligned}
\text{shape}\langle s \rangle &= [(0, 0)] && s \text{ is a scalar} \\
\text{shape}\langle p \rangle &= (d + 1, 0) : (\text{shape } p) && p \text{ is a powerlist of depth } d \\
\text{shape}(p \mid q) &= (d, n + 1) : z && \text{where } (\text{shape } p) = (d, n) : z \\
\text{shape}(p \bowtie q) &= (d, n + 1) : z && \text{where } (\text{shape } p) = (d, n) : z
\end{aligned}$$

Two powerlists are *similar* if they have identical shape and types of elements. Powerlist constructors, $\mid^m$ and $\bowtie^n$, are applied only on similar powerlists.

8.6.1.4 Proofs over Multi-dimensional Powerlists

Most proofs over mdp use structural induction on its shape. For mdp u , consider three cases in the proof, (1) $u = \langle x \rangle$ where x is a scalar, (2) $u = \langle p \rangle$ where p is an mdp, and (3) $u = p \mid q$ or $u = p \bowtie q$ where both p and q are mdp.

The proof of Theorem 8.6 in Appendix 8.A.2 (page 506) and of the embedding algorithm of Section 8.6.2.7 (page 498) illustrate the proof technique.

8.6.1.5 Indices of Nodes in a Multi-dimensional Powerlist

Define function idxh for an mdp, analogous to function idx of Section 8.3.2 (page 466) for a simple powerlist. For an mdp p , $(\text{idxh } p)$ is an mdp of the same shape as p and its elements are indices of the corresponding elements of p as bit strings.

$$\begin{aligned}
\text{idxh } \langle x \rangle &= \langle "" \rangle && \text{where } x \text{ is a scalar} \\
\text{idxh } \langle p \rangle &= \langle \text{idxh } p \rangle && \text{where } p \text{ is an mdp} \\
\text{idxh}(p \mid q) &= ('0' :)(\text{idxh } p) \mid ('1' :)(\text{idxh } q)
\end{aligned}$$

8.6.2 Algorithms over Multi-dimensional Powerlists

There are several matrix algorithms, such as LU decomposition, that can be described in recursive style. I show a few small examples of such algorithms in this section.

8.6.2.1 Outer Product

The outer product of vectors A and B is matrix C , where $C[i, j] = A[i] \times B[j]$. Function outp computes the outer product of two powerlists.

$$\begin{aligned}
\text{outp}\langle x \rangle \langle y \rangle &= \langle \langle x \times y \rangle \rangle \\
\text{outp}(p \mid q)(u \mid v) &= ((\text{outp } p \ u) \mid (\text{outp } p \ v)) \mid^1 ((\text{outp } q \ u) \mid (\text{outp } q \ v))
\end{aligned}$$

8.6.2.2 Matrix Transposition

Let τ be a function that transposes matrices. From the definition of a matrix, we have to consider three cases in defining τ .

$$\begin{aligned}
\tau\langle x \rangle &= \langle x \rangle \\
\tau(p \mid q) &= (\tau p) \mid^1 (\tau q) \\
\tau(u \mid^1 v) &= (\tau u) \mid (\tau v)
\end{aligned}$$

The definition of function τ , though straightforward, has introduced the possibility of inconsistency. For a 2×2 matrix, for instance, either of the last two deconstructions apply, and it is not obvious that the same result is obtained independent of the order in which the rules are applied. I show that τ is indeed a function. That is, for a matrix of the form $(p \mid p') \mid^1 (q \mid q')$, which is same as $(p \mid^1 q) \mid (p' \mid^1 q')$, from (HL3) of Theorem 8.6, applications of the last two rules in different order yield the same result.

$$\begin{aligned}
&\tau((p \mid p') \mid^1 (q \mid q')) \\
&= \{\text{apply the last rule}\} \\
&\quad (\tau(p \mid p')) \mid (\tau(q \mid q')) \\
&= \{\text{apply the middle rule}\} \\
&\quad ((\tau p) \mid^1 (\tau p')) \mid ((\tau q) \mid^1 (\tau q'))
\end{aligned}$$

And, applying the rules in the other order:

$$\begin{aligned}
&\tau((p \mid^1 q) \mid (p' \mid^1 q')) \\
&= \{\text{apply the middle rule}\} \\
&\quad \tau(p \mid^1 q) \mid^1 \tau(p' \mid^1 q') \\
&= \{\text{apply the last rule}\} \\
&\quad ((\tau p) \mid (\tau q)) \mid^1 ((\tau p') \mid (\tau q')) \\
&= \{\text{apply (HL3) of Theorem 8.6}\} \\
&\quad ((\tau p) \mid^1 (\tau p')) \mid ((\tau q) \mid^1 (\tau q'))
\end{aligned}$$

Crucial to the above proof is the fact that $\mid$ and $\mid^1$ distribute. This is reminiscent of the “Church–Rosser Property” [see Church 1941] used in term rewriting systems.

It is easy to show that

$$\begin{aligned}
\tau(p \bowtie q) &= (\tau p) \bowtie^1 (\tau q), \text{ and} \\
\tau(u \bowtie^1 v) &= (\tau u) \bowtie (\tau v)
\end{aligned}$$

Exercises 12 and 13 explore some of the properties of function τ .

8.6.2.3 Transposition of Square Matrix

Transpose a square matrix by deconstructing it into quarters, transposing each quarter individually, and re-arranging them, as shown for the transposition function σ in Figure 8.7. Note the effectiveness of pattern matching in this definition.

$$\sigma \begin{bmatrix} p & p' \\ q & q' \end{bmatrix} = \begin{bmatrix} \sigma p & \sigma q \\ \sigma p' & \sigma q' \end{bmatrix}$$

Figure 8.7 Schematic of the transposition of a square matrix.

$$\begin{aligned} \sigma \langle \langle x \rangle \rangle &= \langle \langle x \rangle \rangle \\ \sigma((p \mid p')^\perp (q \mid q')) &= ((\sigma p)^\perp (\sigma p')) \mid ((\sigma q)^\perp (\sigma q')) \end{aligned}$$

8.6.2.4 Square Matrix Multiplication

I develop two algorithms for square matrix multiplication. The simplest algorithm is to quarter each matrix into four submatrices, multiply the submatrices recursively and add the corresponding submatrices, as shown in Figure 8.8 (page 497). Symbols $\oplus$ and $\otimes$ are binary operators for matrix addition and multiplication, respectively.

The definition of $\oplus$ is quite easy, and I omit it. The definition of $\otimes$ is given below.

$$\begin{aligned} \langle \langle x \rangle \rangle \otimes \langle \langle y \rangle \rangle &= \langle \langle x \times y \rangle \rangle \\ ((p \mid p')^\perp (q \mid q')) \otimes ((u \mid u')^\perp (v \mid v')) \\ &= ((p \otimes u) \oplus (p' \otimes v) \mid (p \otimes u') \oplus (p' \otimes v')) \\ &\quad {}^\perp ((q \otimes u) \oplus (q' \otimes v) \mid (q \otimes u') \oplus (q' \otimes v')) \end{aligned}$$

$$\begin{bmatrix} p & p' \\ q & q' \end{bmatrix} \otimes \begin{bmatrix} u & u' \\ v & v' \end{bmatrix} = \begin{bmatrix} p \otimes u \oplus p' \otimes v & p \otimes u' \oplus p' \otimes v' \\ q \otimes u \oplus q' \otimes v & q \otimes u' \oplus q' \otimes v' \end{bmatrix}$$

Figure 8.8 Square matrix multiplication.

8.6.2.5 Strassen's Matrix Multiplication Algorithm

[Strassen \[1969\]](#) has developed an algorithm for matrix multiplication that is asymptotically superior to the simple algorithm, given above. The simple algorithm has a complexity of $\mathcal{O}(N^3)$ whereas Strassen's algorithm is approximately $\mathcal{O}(N^{2.807})$ for $N \times N$ matrices. The improved algorithm outperforms the simple algorithm in practice for N around 1,000; so, the benefits are realized only for very large matrices. There are even better algorithms. [Coppersmith and Winograd \[1990\]](#) propose a sophisticated algorithm that takes $\mathcal{O}(N^{2.376})$ time, but it is effective only for very large matrices.

The simple algorithm involves 8 multiplications and 4 additions of the submatrices that are a quarter of the size of the original matrix. Strassen employs only 7 multiplications though there are more additions. Use the same symbols as in Figure 8.8 for the matrices to be multiplied. First compute 7 matrices, t_i , $1 \leq i \leq 7$.

Below, $\ominus$ is matrix subtraction, which can be implemented in the same way as matrix addition. Observe that computation of each t_i , $1 \leq i \leq 7$, requires one multiplication of submatrices a quarter of the size of the original matrix.

$$\begin{aligned}
 t_1 &= (p \oplus q') \otimes (u \oplus v') \\
 t_2 &= (q \oplus q') \otimes u \\
 t_3 &= p \otimes (u' \ominus v') \\
 t_4 &= q' \otimes (v \ominus u) \\
 t_5 &= (p \oplus p') \otimes v' \\
 t_6 &= (v \ominus u) \otimes (u \oplus u') \\
 t_7 &= (p' \ominus q') \otimes (v \oplus v')
 \end{aligned}$$

Next, compute matrices r , r' , s and s' as shown below. The final result of matrix multiplication is $((r \mid r') \mid^1 (s \mid s'))$.

$$\begin{aligned}
 r &= t_1 \oplus t_4 \ominus t_5 \oplus t_7 & r' &= t_3 \oplus t_5 \\
 s &= t_2 \oplus t_4 & s' &= t_1 \ominus t_2 \oplus t_3 \oplus t_6
 \end{aligned}$$

8.6.2.6 Gray Code in Multi-dimensional Powerlist

The following function *grh* is the generalization of function *gr* of Section 8.4.3.5 (page 471). Applied to an mdp p of 2^n elements, *grh* returns an mdp of n -bit strings of the same shape as p in which adjacent elements along any dimension differ in exactly one bit position. Adjacent pairs in a wraparound fashion along any dimension also differ in a single bit position.

$$\begin{aligned}
 grh \langle x \rangle &= \langle "" \rangle && \text{where } x \text{ is a scalar} \\
 grh \langle p \rangle &= \langle grh p \rangle && \text{where } p \text{ is a powerlist} \\
 grh (p \mid q) &= ('0' :)(grh p) \mid revp(('1' :)(grh q))
 \end{aligned}$$

8.6.2.7 Embedding in Hypercubes

I have introduced *hypercube* in the introductory section of this chapter. To recap, an n -dimensional hypercube is a graph of 2^n nodes, $n \geq 0$, where each node has n neighbors. Therefore, it is possible to assign a unique n -bit index to every node so that the indices of neighbors differ in a single bit.

The data stored at the nodes of an n -dimensional hypercube is an mdp of depth n where any powerlist includes exactly two elements. I generalize this definition slightly to allow a powerlist at any depth to be a singleton. Specifically, a hypercube is either (1) $\langle x \rangle$ where x is a scalar or hypercube or (2) $\langle p \rangle \mid \langle q \rangle$ where p and q are hypercubes. Here, (1) for a scalar x , $\langle x \rangle$ represents a 0-dimensional hypercube, and for a hypercube p , $\langle p \rangle$ is a hypercube whose dimension is 1 more than

that of p , and $(2) \langle p \rangle | \langle q \rangle$ is a $(n + 1)$ -dimensional hypercube that contains two n -dimensional hypercubes, p and q . Note, in particular, that a powerlist like $\langle a b c d \rangle$ is not a hypercube because it has more than two elements along one dimension.

Embedding problem Define two nodes in an mdp to be *neighbors* if they are adjacent along any one dimension, where adjacent means being next to each other in a wraparound fashion. In a simple powerlist like $\langle a b c d \rangle$, each node has exactly two neighbors, a and c for b , and b and d for a . In multi-dimensional powerlist, like $\langle \langle 2 0 3 6 \rangle \langle 4 1 5 7 \rangle \rangle$, 2 has neighbors 0 and 6 along the first dimension, neighbor 4 in the other dimension. As explained earlier, each node in a n -dimensional hypercube has n neighbors.

The embedding theorem says that it is possible to allocate the elements of any mdp to a hypercube, that has the same number of nodes, so that neighbors in the mdp remain neighbors in the hypercube as well. I show an algorithm for embedding and prove its correctness.

The algorithm operates as follows for an mdp of 2^n elements. For each dimension, construct a Gray code sequence and assign the bit strings from the sequence to the nodes along that dimension as a *partial g-index*. A complete *g-index*, or simply *g-index*, is the concatenation of all partial *g-indices* at a node in order of dimension numbers. Let g_i , an n -bit string, be the *g-index* of node with index i . Assign the element at node i of the mdp to node of index g_i in a hypercube of n dimensions. I claim that this embedding preserves neighbors.

Observe that neighbors along a given dimension in the mdp differ in exactly one bit position in their partial *g-index* along that dimension and differ in no other partial *g-index*. Therefore, for neighbors i and j , g_i and g_j differ in exactly one bit position. Since they are assigned to nodes with indices g_i and g_j in the hypercube, they are neighbors in the hypercube, from the definition of indices of a hypercube.

Embedding algorithm The embedding algorithm follows the outline described above except that there is no need to construct the Gray code explicitly. It is a generalization of *pgr* of Section 8.4.3.5 (page 471) for mdp.

$$\begin{aligned} \text{emb } \langle x \rangle &= \langle x \rangle && \text{where } x \text{ is a scalar} \\ \text{emb } \langle p \rangle &= \langle \text{emb } p \rangle && \text{where } p \text{ is a powerlist} \\ \text{emb } (p | q) &= \langle \langle \text{emb } p \rangle (\text{emb } (\text{rev } p)) \rangle \end{aligned}$$

Using induction on the shape of p , the following facts are easily proved. For any powerlist p : (1) $(\text{emb } p)$ is a powerlist, (2) $(\text{emb } p)$ is a permutation of the elements of p , that is, it includes all elements of p and none others, and (3) each powerlist in $(\text{emb } p)$ has either one or two elements. Also, each element of $(\text{emb } p)$ is a powerlist, not a scalar; so, $\text{emb } \langle a b \rangle = \langle \langle a \rangle \langle b \rangle \rangle$. This choice is made so that $\text{emb } \langle a \rangle$

for scalar a , which is $\langle a \rangle$, has depth 0 whereas $emb\langle a\ b \rangle = \langle \langle a \rangle \langle b \rangle \rangle$ has depth 1. The depth of $(emb\ p)$ is n , where p has 2^n elements but includes no singleton powerlist as an element. The depth increases by one for each singleton powerlist, so $emb\langle \langle a \rangle \rangle = \langle \langle a \rangle \rangle$ has depth 1.

Correctness of embedding A proof of correctness is essential for *emb* because it is not intuitively obvious. I use the proof technique for permutation functions (see Section 8.4.2, page 467) but generalize the principle for mdp. I use an index function for an mdp that has the same shape as its argument.

For a multi-dimensional powerlist u , let the index corresponding to an element be its g-index. A correct embedding of u will place every element in the position of its g-index in a hypercube, so the resulting hypercube consists of elements that have the index of the position they occupy. This amounts to proving: $emb(grh\ u) = (idxh\ u)$, for every powerlist u , where *grh* is defined in Section 8.6.2.6 (page 498) and *idxh* in Section 8.6.1.5 (page 495). The proof is by structural induction (see Appendix 8.A.3, page 508).

8.7 Other Work on Powerlist

Powerlists were first described and applied in a number of algorithms in Misra [1994]. Misra [2000] shows how regular interconnection networks, such as butterfly network, can be described by using generating functions over powerlists.

Kapur and Subramaniam [1995, 1998] have proved many of the algorithms in this chapter using the automatic inductive theorem prover RRL (Rewrite Rule Laboratory) that is based on equality reasoning and rewrite rules.

One of the fundamental problems with the powerlist notation is to devise compilation strategies for mapping programs (written in the powerlist notation) to specific architectures. The conceptually closest architecture is the hypercube. Kornerup [1997a, 1997b] has developed certain strategies whereby each parallel step in a program is mapped to a constant number of local operations and communications at a hypercube node.

Divide-and-conquer algorithms often admit massive parallelization and are amenable to implementation using powerlist, as shown in Achatz and Schulte [1996].

Combinational circuit verification is an area in which the powerlist notation may be fruitfully employed. A ripple-carry adder is typically easy to describe and prove whereas a carry-lookahead adder is much more difficult. Adams [1994] has described both circuits in the powerlist notation and proved their equivalence in a remarkably concise fashion.

A generalization of coverset induction, which is used in algebraic specifications, has been used in Hendrix et al. [2010] to reason about powerlist algorithms.

This principle has been implemented in the Maude ITP and used to perform an extensive case study on powerlist algorithms.

8.8 Exercises

1. Write a function over a non-singleton powerlist of integers that computes a powerlist of half the length by adding pairs of adjacent elements. So, with argument powerlist $\langle 3\ 1\ 2\ 6 \rangle$, the function value is $\langle 4\ 8 \rangle$.

Solution $adjsun(u \bowtie v) = u + v$

2. Function *read* returns the value at index i of powerlist p . Code the function; assume that i is given as a bit-string.
3. Prove that $revp(revp\ p) = p$, for any powerlist p .
4. Prove $revp \circ rr = rl \circ revp$, where $\circ$ is function composition.
5. Prove for any powerlist p :

(a) $grr\ 0\ p = p$.

(b) $grr(-1)\ p = rl\ p$.

(c) $(grr\ k\ p) = (rr^{(k)}\ p)$, where $rr^{(k)}$ is the k -fold application of rr , for $k \geq 0$.

(d) $grr\ k\ p = grr\ (k \bmod 2^n)\ p$, where length of p is 2^n .

6. (a) Code function *swapany* that swaps any two elements of a powerlist, not just adjacent elements.

Hint Use the same general idea as in *swap* of Section 8.4.3.2 (page 468). Observe that instead of the last two cases in the definition of *swap*, where the elements to be swapped belong in different halves of the powerlist, you have to consider four cases for swapping x and y in $((ul\ |\ ur)\ |\ (vl\ |\ vr))$: x could be in ul or ur , and y in vl or vr .

- (b) Prove the correctness of *swapany*.

Hint Use function *read* from Exercise 2 to specify *swapany*.

7. Prove the correctness of function *pgr* of Section 8.4.3.5 (page 471).

Hint According to the informal specification, if *pgr* is applied to the powerlist of n -bit indices in Gray code order, that is, $(gray\ n)$, it will permute each element in the Gray code order, placing each index in its right place in ascending order, that is, $(idx\ n)$, for all $n, n \geq 0$. So, the formal definition of correctness is $pgr(gray\ n) = (idx\ n)$, for all $n, n \geq 0$.

See Section 8.6.2.7 (page 498) for the description of a more general function, *emb*, that permutes elements according to Gray code order in a multi-dimensional powerlist. Its proof appears in Appendix 8.A.3 (page 508).

8. Define a boolean-valued function that determines if two powerlists of equal length differ in their corresponding positions exactly once (see Section 7.7 part (6) for a similar function for linear lists).

Solution

$$\begin{aligned} hp \langle x \rangle \langle y \rangle &= (x \neq y) \\ hp (p \mid q) (r \mid s) &= ((hp \ p \ r) \wedge (q = s)) \vee ((p = r) \wedge (hp \ q \ s)) \end{aligned}$$

9. Define boolean-valued function *wadj* over non-singleton powerlists where (*wadj* *u*) is *true* iff every pair of adjacent elements in *u*, including the pair of its first and last element (to account for wraparound adjacency), differ. Thus, *wadj*⟨3 7 3 9⟩ is *true* whereas *wadj*⟨9 7 3 9⟩ is *false*.

Hint Define function *dif* over powerlists *p* and *q* of equal length whose value is *true* iff every pair of corresponding elements in *p* and *q* are different. Observe that for non-singleton powerlist *u*, *wadj* *u* = *dif* *u* (*rr* *u*), where *rr* is the right-rotate function defined in Section 8.4.3.1 (page 468).

Solution

$$\begin{aligned} dif \langle x \rangle \langle y \rangle &= (x \neq y) \\ dif (p \bowtie q) (r \bowtie s) &= (dif \ p \ r) \wedge (dif \ q \ s) \end{aligned}$$

You can compute a direct definition of *wadj* for non-singleton argument (*p* $\bowtie$ *q*),

$$\begin{aligned} &wadj(p \bowtie q) \\ &= \{wadj \ u = dif \ u \ (rr \ u), \text{ substitute } (p \bowtie q) \text{ for } u\} \\ &\quad dif(p \bowtie q) (rr \ p \bowtie q) \\ &= \{(rr \ p \ bowtie q) = (rr \ q) \ bowtie p\} \\ &\quad dif(p \ bowtie q) ((rr \ q) \ bowtie p) \\ &= \{dif \text{ is a pointwise function}\} \\ &\quad (dif \ p \ (rr \ q)) \wedge (dif \ q \ p) \\ &= \{\text{rearrange terms; } dif \text{ and } \wedge \text{ are symmetric in their arguments}\} \\ &\quad (dif \ p \ q) \wedge (dif \ p \ (rr \ q)) \end{aligned}$$

10. Prove the following properties of bitonic 0–1 powerlists.

- (a) $\langle x \rangle$ is bitonic,
- (b) $(p \bowtie q)$ bitonic $\Rightarrow p$ bitonic $\wedge q$ bitonic $\wedge |zp - zq| \leq 1$,
- (c) p sorted, q sorted $\Rightarrow (p \mid (revp \ q))$ bitonic.

Hint For proofs about bitonic 0–1 powerlists, write such a list as concatenation of three segments, $a \ ++ \ b \ ++ \ c$, where *a* is all 0's, *b* is all 1's and *c* is all

0's; one or two of the segments may be empty. (A similar analysis applies if a is all 1's, b is all 0's and c is all 1's.) A possible schematic of $p \bowtie q$ as a bitonic powerlist is, writing p for each occurrence of an element of p :

$$\overbrace{pqpqpqpqpqpqpqpq}^{\begin{smallmatrix} 0's & 1's & 0's \end{smallmatrix}}.$$

Solution Proofs of (10a) and (10c) are straightforward. I prove (10b).

Let $p \bowtie q$ be a bitonic powerlist of the form $a \uplus b \uplus c$, where a is all 0's, b is all 1's and c is all 0's. It is easy to see that p is of the form $a' \uplus b' \uplus c'$, where a' , b' and c' are subsequences of a , b and c , respectively. So, a' is all 0's, b' is all 1's and c' is all 0's; hence, p is bitonic. Similarly, q is bitonic.

To compare the number of zeroes of p and q , observe that any segment of even length (call it “even segment”) contributes equal number of 0's to both p and q , and an odd segment of 0's contributes one more zero to p if it starts with p , to q otherwise. In all cases, the number of 0's in p and q from a single segment differ by at most one. Since the length of $p \bowtie q$ is even and there are three segments, not all three segments are odd, that is, there is at least one even segment (possibly of length 0). If a is even, then $|zp - zq| \leq 1$ because all zeroes come from c ; similarly if c is even. If b is even, then $p \bowtie q$ is all zeroes, so $zp = zq$.

11. For prefix sum, $psum$ (defined in Section 8.5.5, page 485), show that

$$psum(u \bowtie v)^{\rightarrow} = (x \oplus v)^{\rightarrow} \bowtie x, \text{ where } x = psum(v^{\rightarrow} \oplus u).$$

Hint Use Theorem 8.5 (page 487).

12. For function τ (defined in Section 8.6.2.2, page 495), prove that for powerlists p and q :

$$(a) \quad \tau(\tau p) = p$$

$$(b) \quad \tau(p \mid^1 q) = (\tau p) \mid (\tau q)$$

$$(c) \quad \tau(p \mid q) = (\tau p) \mid^1 (\tau q)$$

13. Prove that function τ (defined in Section 8.6.2.2, page 495) actually transposes matrices. Observe that τ is a permutation function, so apply the techniques described in Section 8.4.4 (page 472).

Hint Let function $rcidx$ assign the pair of row and column index to each position in a matrix, so for the matrix that is given column-wise, $rcidx \langle \langle 2 \ 3 \rangle \langle 4 \ 5 \rangle \rangle = \langle \langle (0, 0) \ (1, 0) \rangle \langle (0, 1) \ (1, 1) \rangle \rangle$. The scalar function $flip$ applied to a pair (x, y) returns (y, x) . For any matrix p , show that $\tau(rcidx p) = flip(rcidx(\tau p))$, that is, row and column indices are transposed.

8.A Appendices

Notational devices Many of the proofs in this chapter include expressions with multiple levels of parentheses. Traditional mathematics addresses this problem by introducing binding rules so that some of the parentheses can be removed. Computer scientists have used other techniques, such as two-dimensional display, as in Haskell, in which parentheses are aligned in different lines to show their scope. It is also possible to assign subscripts to parentheses pairs to make it easier to see the scope, as in $(_0 3 \times (_1 4 + 6 _1)_0)$.

I introduce some additional devices for parentheses elimination: use of colors, multiple typefaces and overline, made possible by advances in typesetting technology and color displays and printers. A string of symbols that are in different color or typeface (font) from its surrounding symbols or has a straight-line over it should be treated as if it is enclosed within parentheses. For example,

$\overline{\text{rev inv '0': idx n} \mid \text{'1': idx n}}$ represents
 $\text{rev}(\text{inv } (('0' :)(\text{idx } n) \mid ('1' :)(\text{idx } n)))$.

I show the proof of function *inv* in the inductive case, first in traditional style, next after parentheses elimination. Unfortunately, the colors cannot be appreciated on a monochrome display or on a black and white page.

8.A.1 Proof of Correctness of *inv*

Function *inv* is one of the harder permutation functions to prove (or even understand). Applied to $(\text{idx } n)$, it reverses the bit string at each position of the powerlist. Therefore, reversing each bit string of $(\text{inv } (\text{idx } n))$ should yield $(\text{idx } n)$. We get the specification:

$$\text{rev}(\text{inv } (\text{idx } n)) = (\text{idx } n), \text{ for all } n, n \geq 0.$$

I choose a simple definition of *rev* that is easy to use in a proof:

$$\begin{aligned} \text{rev } [] &= [] \\ \text{rev } (x : xs) &= (\text{rev } xs) ++ [x] \end{aligned}$$

- $n = 0, \text{rev}(\text{inv } (\text{idx } 0)) = (\text{idx } 0)$:

$$\begin{aligned} &\text{rev}(\text{inv } (\text{idx } 0)) \\ &= \{\text{definition of idx}\} \\ &\quad \text{rev}(\text{inv } \langle "" \rangle) \\ &= \{\text{definition of inv}\} \\ &\quad \text{rev} \langle "" \rangle \\ &= \{\text{definition of rev}\} \end{aligned}$$

$$\begin{aligned}
& \langle "" \rangle \\
= & \{\text{definition of } idx\} \\
& idx\ 0
\end{aligned}$$

• $n + 1, rev(inv\ (idx\ (n + 1))) = (idx\ (n + 1))$: This is a longer proof because there are three functions, rev, inv, idx , in the proof. The proof uses the following scalar functions: $('0' :)$ and $('1' :)$ that append a '0' and '1', respectively, at the beginning of a string, and $(++\ "0")$ and $(++\ "1")$ that append at the end.

• $n + 1, rev(inv\ (idx\ (n + 1))) = (idx\ (n + 1))$:

$$\begin{aligned}
& rev(inv\ (idx\ (n + 1))) \\
= & \{\text{definition of } idx\} \\
& rev(inv\ (('0' :)(idx\ n) \mid ('1' :)(idx\ n))) \\
= & \{\text{definition of } inv\} \\
& rev(inv\ (('0' :)(idx\ n)) \bowtie (inv\ (('1' :)(idx\ n))) \\
= & \{\text{scalar function } rev \text{ distributes over } \bowtie\} \\
& rev(inv\ ('0' :)(idx\ n)) \bowtie rev(inv\ ('1' :)(idx\ n)) \\
= & \{\text{permutation function } inv \text{ distributes over scalar function } rev\} \\
& inv(rev\ ('0' :)(idx\ n)) \bowtie inv(rev\ ('1' :)(idx\ n)) \\
= & \{\text{definition of } rev\} \\
& inv((++\ "0")(rev(idx\ n))) \bowtie inv((++\ "1")(rev(idx\ n))) \\
= & \{inv \text{ distributes over scalar functions } (++\ "0") \text{ and } (++\ "1")\} \\
& (++\ "0")(inv(rev(idx\ n))) \bowtie (++\ "1")(inv(rev(idx\ n))) \\
= & \{\text{permutation function } inv \text{ distributes over scalar function } rev\} \\
& ((++\ "0")inv(rev((idx\ n)))) \bowtie ((++\ "1")inv(rev((idx\ n)))) \\
= & \{\text{induction: } rev(inv(idx\ n)) = idx\ n\} \\
& ((++\ "0")(idx\ n)) \bowtie ((++\ "1")(idx\ n)) \\
= & \{\text{property of } idx\} \\
& idx\ (n + 1)
\end{aligned}$$

Eliminating parentheses using new notational devices

$$\begin{aligned}
& rev(inv\ (idx(n + 1))) \\
= & \{\text{eliminating parentheses}\} \\
& \overline{\overline{rev\ inv\ idx\ n + 1}} \\
= & \{\text{definition of } idx\} \\
& \overline{\overline{rev\ inv\ '0':idx\ n \mid '1':idx\ n}} \\
= & \{\text{definition of } inv\}
\end{aligned}$$

$$\begin{aligned}
& \overline{\overline{rev \text{ inv } '0': idx \ n} \bowtie \overline{\overline{inv \text{ '1': idx \ n}}}} \\
= & \{ \text{scalar function } rev \text{ distributes over } \bowtie \} \\
& \overline{\overline{rev \text{ inv } '0': idx \ n} \bowtie \overline{\overline{rev \text{ inv } '1': idx \ n}}} \\
= & \{ \text{permutation function } inv \text{ distributes over scalar function } rev \} \\
& \overline{\overline{inv \text{ rev } '0': idx \ n} \bowtie \overline{\overline{inv \text{ rev } '1': idx \ n}}} \\
= & \{ \text{definition of } rev \} \\
& \overline{\overline{inv \text{ ++"0" rev idx \ n} \bowtie \overline{\overline{inv \text{ ++"1" rev idx \ n}}}}} \\
= & \{ inv \text{ distributes over scalar functions ++"0" and ++"1" } \} \\
& \overline{\overline{++"0" inv \text{ rev idx \ n} \bowtie \overline{\overline{++"1" inv \text{ rev idx \ n}}}}} \\
= & \{ \text{permutation function } inv \text{ distributes over scalar function } rev \} \\
& \overline{\overline{++"0" rev \text{ inv idx \ n} \bowtie \overline{\overline{++"1" rev \text{ inv idx \ n}}}}} \\
= & \{ \text{induction: } rev \text{ inv idx \ n} = idx \ n \} \\
& \overline{\overline{++"0" idx \ n \bowtie \overline{\overline{++"1" idx \ n}}}} \\
= & \{ \text{property of } idx \} \\
& idx \ (n + 1)
\end{aligned}$$

8.A.2 Proofs of Laws About Higher-dimensional Constructors

Theorem 8.6 (page 494) has several parts. Their proofs follow the same pattern, induction on the index k of a constructor, $|^k$ or $\bowtie^k$, along with induction on the length of the argument powerlists. The proofs are long because there are different indices on different constructors. I prove one of the harder results:

$$(p \mid^m p') \mid^k (q \mid^m q') = (p \mid^k q) \mid^m (p' \mid^k q'), \text{ where } m \neq k.$$

Proof. The main proof is by induction on m . Each part of the inductive proof then applies induction on the lengths of the argument powerlists.

- Base case, $m = 0$: Setting $m = 0$, we have $\mid^m = |$, so the result to be proved is: $(p \mid p') \mid^k (q \mid q') = (p \mid^k q) \mid (p' \mid^k q')$, where $k > 0$. Proof is by induction on the length of p .

Case 1) The argument lists are singletons, say $p, q = \langle u \rangle, \langle v \rangle$, and $p', q' = \langle u' \rangle, \langle v' \rangle$. The left side of the identity is:

$$\begin{aligned}
& (p \mid p') \mid^k (q \mid q') \\
= & \{ p, q, p', q' = \langle u \rangle, \langle v \rangle, \langle u' \rangle, \langle v' \rangle \} \\
& \langle \langle u \rangle \mid \langle u' \rangle \rangle \mid^k \langle \langle v \rangle \mid \langle v' \rangle \rangle \\
= & \{ \text{Law L0: } \langle u \rangle \mid \langle u' \rangle = \langle u \rangle \bowtie \langle u' \rangle \} \\
& \langle \langle u \rangle \bowtie \langle u' \rangle \rangle \mid^k \langle \langle v \rangle \bowtie \langle v' \rangle \rangle \\
= & \{ \bowtie \text{ and } \mid^k \text{ distribute, from the definition of } \mid^k \}
\end{aligned}$$

$$\begin{aligned}
& (\langle u \rangle |^k \langle v \rangle) \bowtie (\langle u' \rangle |^k \langle v' \rangle) \\
= & \{\text{for } k > 0, \langle u \rangle |^k \langle v \rangle \text{ is a singleton. Law L0}\} \\
& (\langle u \rangle |^k \langle v \rangle) | (\langle u' \rangle |^k \langle v' \rangle) \\
= & \{p, q = \langle u \rangle, \langle v \rangle, p', q' = \langle u' \rangle, \langle v' \rangle\} \\
& (p |^k q) | (p' |^k q')
\end{aligned}$$

Case 2) The argument lists are non-singletons, say

$$p, q = u \bowtie v, r \bowtie s \text{ and } p', q' = u' \bowtie v', r' \bowtie s'.$$

The result to be proved is:

$$\begin{aligned}
& ((u \bowtie v) | (u' \bowtie v')) |^k ((r \bowtie s) | (r' \bowtie s')) \\
= & ((u \bowtie v) |^k (r \bowtie s)) | ((u' \bowtie v') |^k (r' \bowtie s')), \text{ where } k > 0.
\end{aligned}$$

The left side of the identity is:

$$\begin{aligned}
& ((u \bowtie v) | (u' \bowtie v')) |^k ((r \bowtie s) | (r' \bowtie s')) \\
= & \{ | \text{ and } \bowtie \text{ distribute} \} \\
& ((u | u') \bowtie (v | v')) |^k ((r | r') \bowtie (s | s')) \\
= & \{\text{from the definition of } |^k, \bowtie \text{ and } |^k \text{ distribute}\} \\
& ((u | u') |^k (r | r')) \bowtie ((v | v') |^k (s | s')) \\
= & \{ | \text{ and } |^k \text{ distribute, by induction on argument powerlist lengths} \} \\
& ((u |^k r) | (u' |^k r')) \bowtie ((v |^k s) | (v' |^k s')) \\
= & \{ | \text{ and } \bowtie \text{ distribute} \} \\
& ((u |^k r) \bowtie (v |^k s)) | ((u' |^k r') \bowtie (v' |^k s')) \\
= & \{ \bowtie \text{ and } |^k \text{ distribute, from the definition of } |^k \} \\
& ((u \bowtie v) |^k (r \bowtie s)) | ((u' \bowtie v') |^k (r' \bowtie s'))
\end{aligned}$$

• Inductive case, $m > 0$: The required proof obligation is:

$$(p |^k p') |^m (q |^k q') = (p |^m q) |^k (p' |^m q'), \text{ where } m \neq k.$$

Proof is by induction on the lengths of argument powerlists. For $k = 0$, the proof for the base case applies, using m in place of k . So, assume that $k > 0$.

Case 1) All arguments are singletons: $p, q = \langle u \rangle, \langle v \rangle$ and $p', q' = \langle u' \rangle, \langle v' \rangle$. Observe that for singleton powerlists $\langle r \rangle$ and $\langle s \rangle$ and $n > 0$, $\langle r \rangle |^n \langle s \rangle = \langle r |^{n-1} s \rangle$ is a singleton powerlist.

$$\begin{aligned}
& (p |^k p') |^m (q |^k q') \\
= & \{p, q, p', q' = \langle u \rangle, \langle v \rangle, \langle u' \rangle, \langle v' \rangle\} \\
& (\langle u \rangle |^k \langle u' \rangle) |^m (\langle v \rangle |^k \langle v' \rangle) \\
= & \{\text{definition of } |^k \text{ over singletons, } k > 0\} \\
& \langle u |^{k-1} u' \rangle |^m \langle v |^{k-1} v' \rangle \\
= & \{\text{definition of } |^m \text{ over singletons, } m > 0\}
\end{aligned}$$

$$\begin{aligned}
& \langle (u \mid^{k-1} u') \mid^{m-1} (v \mid^{k-1} v') \rangle \\
= & \{ \text{Inductively, } \mid^{k-1} \text{ and } \mid^{m-1} \text{ distribute} \} \\
& \langle (u \mid^{m-1} v) \mid^{k-1} (u' \mid^{m-1} v') \rangle \\
= & \{ \text{definition of } \mid^k \text{ over singletons, } k > 0 \} \\
& \langle u \mid^{m-1} v \rangle \mid^k \langle u' \mid^{m-1} v' \rangle \\
= & \{ \text{definition of } \mid^m \text{ over singletons, } m > 0 \} \\
& (\langle u \rangle \mid^m \langle v \rangle) \mid^k (\langle u' \rangle \mid^m \langle v' \rangle) \\
= & \{ p, q = \langle u \rangle, \langle v \rangle, p', q' = \langle u' \rangle, \langle v' \rangle \} \\
& (p \mid^m q) \mid^k (p' \mid^m q')
\end{aligned}$$

Case 2) The argument lists are non-singletons, say

$p, q = u \bowtie v, r \bowtie s$ and $p', q' = u' \bowtie v', r' \bowtie s'$.

The result to be proved, given $m > 0, k > 0$, and $m \neq k$:

$$\begin{aligned}
& ((u \bowtie v) \mid^k (u' \bowtie v')) \mid^m ((r \bowtie s) \mid^k (r' \bowtie s')) \\
= & ((u \bowtie v) \mid^m (r \bowtie s)) \mid^k ((u' \bowtie v') \mid^m (r' \bowtie s')).
\end{aligned}$$

The left side of the identity is:

$$\begin{aligned}
& ((u \bowtie v) \mid^k (u' \bowtie v')) \mid^m ((r \bowtie s) \mid^k (r' \bowtie s')) \\
= & \{ \text{from the definition of } \mid^k, \bowtie \text{ distributes over } \mid^k \} \\
& ((u \mid^k u') \bowtie (v \mid^k v')) \mid^m ((r \mid^k r') \bowtie (s \mid^k s')) \\
= & \{ \text{from the definition of } \mid^m, \bowtie \text{ distributes over } \mid^m \} \\
& ((u \mid^k u') \mid^m (r \mid^k r')) \bowtie ((v \mid^k v') \mid^m (s \mid^k s')) \\
= & \{ \mid^k \text{ and } \mid^m \text{ distribute, by induction on the lengths} \\
& \text{of the arguments} \} \\
& ((u \mid^m r) \mid^k (u' \mid^m r')) \bowtie ((v \mid^m s) \mid^k (v' \mid^m s')) \\
= & \{ \text{from the definition of } \mid^k, \bowtie \text{ distributes over } \mid^k \} \\
& ((u \mid^m r) \bowtie (v \mid^m s)) \mid^k ((u' \mid^m r') \bowtie (v' \mid^m s')) \\
= & \{ \text{from the definition of } \mid^m, \bowtie \text{ distributes over } \mid^m \} \\
& ((u \bowtie v) \mid^m (r \bowtie s)) \mid^k ((u' \bowtie v') \mid^m (r' \bowtie s'))
\end{aligned}$$

8.A.3 Proof of Hypercube Embedding Algorithm

I repeat the definitions of three functions, *idxh*, *grh* and *emb*, here for easy reference.

$$\begin{aligned}
idxh \langle x \rangle &= \langle "" \rangle && \text{where } x \text{ is a scalar} \\
idxh \langle p \rangle &= \langle idxh \, p \rangle && \text{where } p \text{ is an mdp} \\
idxh(p \mid q) &= '0':(idxh \, p) \mid '1':(idxh \, q) \\
grh \langle x \rangle &= \langle "" \rangle && \text{where } x \text{ is a scalar} \\
grh \langle p \rangle &= \langle grh \, p \rangle && \text{where } p \text{ is a powerlist} \\
grh(p \mid q) &= '0':(grh \, p) \mid revp('1':(grh \, q))
\end{aligned}$$

$$\begin{aligned}
emb \langle x \rangle &= \langle x \rangle && \text{where } x \text{ is a scalar} \\
emb \langle p \rangle &= \langle emb \, p \rangle && \text{where } p \text{ is a powerlist} \\
emb (p \mid q) &= \langle emb \, p \rangle \mid \langle emb (revp \, q) \rangle
\end{aligned}$$

The proof obligation is $emb(grh \, u) = (idxh \, u)$. The proof uses structural induction on u .

- Base case, $u = \langle x \rangle$ for scalar x :

$$\begin{aligned}
&emb(grh \, \langle x \rangle) \\
= &\{\text{definition of } grh\} \\
&emb \langle "" \rangle \\
= &\{\text{definition of } emb\} \\
&\langle "" \rangle \\
= &\{\text{definition of } idxh\} \\
&idxh \langle x \rangle
\end{aligned}$$

- Inductive case, $u = \langle p \rangle$ for powerlist p :

$$\begin{aligned}
&emb(grh \, \langle p \rangle) \\
= &\{\text{definition of } grh\} \\
&emb \langle grh \, p \rangle \\
= &\{\langle grh \, p \rangle \text{ is a singleton powerlist of a powerlist}\} \\
&\langle emb \, grh \, p \rangle \\
= &\{\text{induction: } shape(p), \text{ hence } shape(grh \, p), \text{ included in } shape(u)\} \\
&\langle idxh \, p \rangle \\
= &\{\text{definition of } idxh\} \\
&idxh \langle p \rangle
\end{aligned}$$

- Inductive case, $u = p \mid q$: Note that emb , being a permutation function, distributes over scalar functions, ('0':) and ('1':).

$$\begin{aligned}
&\overline{emb \, grh \, p \mid q} \\
= &\{\text{definition of } grh\} \\
&\overline{emb \, '0': (grh \, p) \mid revp \, '1': (grh \, q)} \\
= &\{\text{definition of } emb\} \\
&\overline{emb \, '0': (grh \, p) \mid emb \, revp (revp \, '1': (grh \, q))} \\
= &\{revp(revp \, r) = r\} \\
&\overline{emb \, '0': (grh \, p) \mid emb \, '1': (grh \, q)} \\
= &\{emb \text{ distributes over '0': and '1':}\}
\end{aligned}$$

$$\begin{aligned}
 & \text{'0': } (emb \overline{grh \ p}) \mid \text{'1': } (emb \overline{grh \ q}) \\
 = & \ \{\text{induction: } (emb \overline{grh \ p}) = (idxh \ p), (emb \overline{grh \ q}) = (idxh \ q)\} \\
 & \text{'0': } (idxh \ p) \mid \text{'1': } (idxh \ q) \\
 = & \ \{\text{definition of } idxh\} \\
 & \ idxh(p \mid q)
 \end{aligned}$$

Bibliography

- K. Achatz and W. Schulte. 1996. Massive parallelization of divide-and-conquer algorithms over powerlists. *Sci. Comput. Program.* 26, 1, 59–78. DOI: [https://doi.org/10.1016/0167-6423\(95\)00022-4](https://doi.org/10.1016/0167-6423(95)00022-4).
- W. Adams. 1994. *Verifying Adder Circuits Using Powerlists*. Technical Report TR 94-02. Department of Computer Science, University of Texas at Austin.
- B. Aspvall, M. F. Plass, and R. E. Tarjan. 1979. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.* 8, 3, 121–123. DOI: [https://doi.org/10.1016/0020-0190\(79\)90002-4](https://doi.org/10.1016/0020-0190(79)90002-4).
- S. K. Basu and J. Misra. 1975. Proving loop programs. *IEEE Trans. Softw. Eng.* SE-1, 1, 76–86. DOI: <https://doi.org/10.1109/TSE.1975.6312822>.
- K. Batchner. 1968. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, Vol. 32. AFIPS Press, Reston, VA, 307–314. DOI: <https://doi.org/10.1145/1468075.1468121>.
- E. Beckenbach and R. Bellman. 1961. *An Introduction to Inequalities*. Mathematical Association of America.
- R. Bellman. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ.
- R. Bellman. 1958. On a routing problem. *Q. Appl. Math.* 16, 1, 87–90. DOI: <https://doi.org/10.1090/qam/102435>.
- J. L. Bentley and M. D. McIlroy. 1993. Engineering a sort function. *Softw. Pract. Exp.* 23, 11, 1249–1265. DOI: <https://doi.org/10.1002/spe.4380231105>.
- R. S. Bird. 2006. Improving saddleback search: A lesson in algorithm design. In *Mathematics of Program Construction, MPC 2006*. Lecture Notes in Computer Science, Vol. 4014. Springer, Berlin, 82–89. DOI: https://doi.org/10.1007/11783596_8.
- B. H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7, 422–426. DOI: <https://doi.org/10.1145/362686.362692>.
- M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7, 4, 448–461. DOI: [https://doi.org/10.1016/S0022-0000\(73\)80033-9](https://doi.org/10.1016/S0022-0000(73)80033-9).
- R. Boyer and J. Moore. 1991. MJRTY—A fast majority vote algorithm. In R. Boyer (Ed.), *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Automated Reasoning Series,

- Vol. 1. Kluwer Academic Publishers, Dordrecht, The Netherlands, 105–117. DOI: https://doi.org/10.1007/978-94-011-3488-0_5.
- R. S. Boyer and J. S. Moore. 1977. A fast string searching algorithm. *Commun. ACM* 20, 10, 762–772. DOI: <https://doi.org/10.1145/359842.359859>.
- R. E. Bryant. 1992. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Comput. Surv.* 24, 3, 293–318. DOI: <https://doi.org/10.1145/136035.136043>.
- R. M. Burstall. 1969. Proving properties of programs by structural induction. *Comput. J.* 12, 1, 41–48. The British Computer Society. DOI: <https://doi.org/10.1093/comjnl/12.1.41>.
- K. M. Chandy and J. Misra. 1984. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.* 6, 4, 632–646.
- M. Charikar, K. Chen, and M. Farach-Colton. 2004. Finding frequent items in data streams. *Theor. Comput. Sci.* 312, 1, 3–15. DOI: [https://doi.org/10.1016/S0304-3975\(03\)00400-6](https://doi.org/10.1016/S0304-3975(03)00400-6).
- A. Church. 1941. *The Calculi of Lambda Conversion*. Princeton University Press.
- E. Clarke and E. Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen (Ed.), *Logics of Programs*. Lecture Notes in Computer Science, Vol. 131. Springer-Verlag, 52–71. DOI: <https://doi.org/10.1007/BFb0025774>.
- S. Cook. 1970. *Linear Time Simulation of Deterministic Two-Way Pushdown Automata*. Technical report. Department of Computer Science, University of Toronto.
- J. M. Cooley and J. W. Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* 19, 90, 297–301. DOI: <https://doi.org/10.1090/S0025-5718-1965-0178586-1>.
- D. Coppersmith and S. Winograd. 1990. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.* 9, 3, 251–280. DOI: [https://doi.org/10.1016/S0747-7171\(08\)80013-2](https://doi.org/10.1016/S0747-7171(08)80013-2).
- H. S. M. Coxeter. 1961. *Introduction to Geometry*. John Wiley & Sons, New York.
- O. Dahl, E. Dijkstra, and C. Hoare. 1972. *Structured Programming*. Academic Press.
- M. Davis and H. Putnam. 1960. A computing procedure for quantification theory. *J. ACM* 7, 3, 201–215. DOI: <https://doi.org/10.1145/321033.321034>.
- M. Davis, G. Logemann, and D. Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7, 394–397. DOI: <https://doi.org/10.1145/368273.368557>.
- N. Dershowitz and Z. Manna. 1979. Proving termination with multiset ordering. *Commun. ACM* 22, 8, 465–476. DOI: <https://doi.org/10.1145/359138.359142>.
- E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1, 269–271. DOI: <https://doi.org/10.1007/BF01386390>.
- E. W. Dijkstra. 1975. Guarded commands, nondeterminacy and the formal derivation of programs. *Commun. ACM* 18, 8, 453–457. DOI: <https://doi.org/10.1145/360933.360975>.
- E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.
- E. W. Dijkstra. May. 1980. A short proof of one of Fermat's theorems. EWD740: Circulated privately. <http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD740.PDF>.
- E. W. Dijkstra. March. 1991. The undeserved status of the pigeon-hole principle. EWD 1094: Circulated privately. <http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1094.PDF>.

- E. W. Dijkstra. September. 1993. A bagatelle on Euclid's algorithm. EWD1158: Circulated privately. <https://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1158.PDF>.
- E. W. Dijkstra. August. 1995. For the record: Painting the squared plane. EWD 1212: Circulated privately. <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1212.html>.
- E. W. Dijkstra. February. 1996a. The arithmetic and geometric means once more. EWD1231: Circulated privately. <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1231.html>.
- E. W. Dijkstra. October. 1996b. Homework #1. EWD1248: Circulated privately. <http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1248.PDF>.
- E. W. Dijkstra. November. 1999. Constructing the binary search once more. EWD1293: Circulated privately. <https://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1293.PDF>.
- E. W. Dijkstra. April. 2002. Coxeter's rabbit. EWD1318: Circulated privately. <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1318.html>.
- E. W. Dijkstra. June. 2003. A problem solved in my head. EWD666: Circulated privately. <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD666.html>.
- E. W. Dijkstra and J. Misra. 2001. Designing a calculational proof of Cantor's theorem. *Am. Math. Mon.* 108, 5, 440–443. DOI: <https://doi.org/10.2307/2695799>.
- E. W. Dijkstra and C. S. Scholten. 1989. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag.
- M. Fitting. 1990. First-order logic. In *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer, 97–125. DOI: https://doi.org/10.1007/978-1-4684-0357-2_5.
- R. W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6, 345. DOI: <https://doi.org/10.1145/367766.368168>.
- R. W. Floyd. 1964. Algorithm 245: Treesort. *Commun. ACM* 7, 12, 701. DOI: <https://doi.org/10.1145/355588.365103>.
- R. W. Floyd. 1967. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics XIX*. American Mathematical Society, 19–32.
- L. R. Ford Jr. 1956. *Network Flow Theory*. Technical report. RAND Corporation, Santa Monica, CA. <https://www.rand.org/pubs/papers/P923>.
- L. Ford, Jr and D. Fulkerson. 1962. *Flows in Networks*. Princeton University Press.
- M. Fredman and M. Saks. 1989. The cell probe complexity of dynamic data structures. In *Proc. of the Twenty-First Annual ACM Symposium on Theory of Computing*. ACM, 345–354. DOI: <https://doi.org/10.1145/73007.73040>.
- M. Fürer. 2007. Faster integer multiplication. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing, STOC'07*. ACM, 57–66. DOI: <https://doi.org/10.1145/1250790.1250800>.
- M. Gardner. 2001. *The Colossal Book of Mathematics: Classic Puzzles, Paradoxes, and Problems: Number Theory, Algebra, Geometry, Probability, Topology, Game Theory, Infinity, and Other Topics of Recreational Mathematics*. WW Norton & Company.

- D. Gale and L. S. Shapley. 1962. College admissions and the stability of marriage. *Am. Math. Monthly* 69, 1, 9–15. DOI: <https://doi.org/10.1080/00029890.1962.11989827>.
- A. V. Goldberg and R. E. Tarjan. 1988. A new approach to the maximum-flow problem. *J. ACM* 35, 4, 921–940. DOI: <https://doi.org/10.1145/48014.61051>.
- D. Gries. 1981. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag. DOI: <https://doi.org/10.1007/978-1-4612-5983-1>.
- D. Gries and F. B. Schneider. 1994. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer-Verlag. DOI: <https://doi.org/10.1007/978-1-4757-3837-7>.
- A. Gupta and J. Misra. 2003. *Synthesizing Programs Over Recursive Data Structures*. Technical Report TR03-38. Computer Science Dept., University of Texas at Austin.
- P. Hall. 1935. On representatives of subsets. *J. Lond. Math. Soc.* 10, 1, 26–30. DOI: <https://doi.org/10.1112/jlms/s1-10.37.26>.
- E. C. R. Hehner. 1984. *The Logic of Programming*. Prentice-Hall. Series in Computer Science, C.A.R. Hoare, series editor.
- E. C. Hehner. 1993. *A Practical Theory of Programming*. Springer-Verlag.
- E. C. R. Hehner. 2022. *A Practical Theory of Programming*. <http://www.cs.utoronto.ca/~hehner/aPToP>. Manuscript available for online access and download.
- J. Hendrix, D. Kapur, and J. Meseguer. 2010. Coverset induction with partiality and subsorts: A powerlist case study. In M. Kaufmann and L. C. Paulson (Eds.), *Interactive Theorem Proving*, Lecture Notes in Computer Science, Vol. 6172. Springer, Berlin, 275–290. ISBN 978-3-642-14052-5. DOI: https://doi.org/10.1007/978-3-642-14052-5_20.
- M. J. H. Heule, O. Kullmann, and V. W. Marek. 2016. Solving and verifying the Boolean Pythagorean Triples problem via cube-and-conquer. In *Proceedings of SAT 2016*. Springer. 228–245. DOI: https://doi.org/10.1007/978-3-319-40970-2_15.
- A. Hinkis. 2013. *Proofs of the Cantor-Bernstein Theorem*. Springer. DOI: <https://doi.org/10.1007/978-3-0348-0224-6>.
- C. A. R. Hoare. 1961. Partition: Algorithm 63, Quicksort: Algorithm 64, and Find: Algorithm 65. *Commun. ACM* 4, 7, 321–322.
- C. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 576–580, 583. DOI: <https://doi.org/10.1145/363235.363259>.
- C. A. R. Hoare and J. Misra. 2009. Preface to the special issue on software verification. *ACM Comput. Surv.* 41, 4, 1–3. DOI: <https://doi.org/10.1145/1592434.1592435>.
- P. Hudak, J. Peterson, and J. Fasel. 2000. A Gentle Introduction to Haskell, Version 98. <http://www.haskell.org/tutorial/>.
- D. Kapur and M. Subramaniam. 1995. Automated reasoning about parallel algorithms using powerlists. In *International Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, Vol. 936. Springer, 416–430. DOI: https://doi.org/10.1007/3-540-60043-4_68.
- D. Kapur and M. Subramaniam. 1998. Mechanical verification of adder circuits using rewrite rule laboratory. *Form. Methods Syst. Des.* 13, 2, 127–158. DOI: <https://doi.org/10.1023/A:1008610818519>.

- V. King, S. Rao, and R. Tarjan. 1994. A faster deterministic maximum flow algorithm. *J. Algorithms* 17, 3, 447–474. DOI: <https://doi.org/10.1006/jagm.1994.1044>.
- D. Kitchin, A. Quark, and J. Misra. 2010. Quicksort: Combining concurrency, recursion, and mutable data structures. In A. W. Roscoe, C. B. Jones, and K. Wood (Eds.), *Reflections on the Work of C.A.R. Hoare*, History of Computing. Written in honor of Sir Tony Hoare's 75th birthday. Springer, 229–254. DOI: https://doi.org/10.1007/978-1-84882-912-1_11.
- D. E. Knuth. 1998. *The Art of Computer Programming* (2nd ed.) Sorting and Searching, Vol. 3. Addison Wesley Longman Publishing.
- D. E. Knuth. 2014. *Art of Computer Programming*, Vol. 2. Seminumerical Algorithms. Addison-Wesley Professional.
- D. E. Knuth, J. H. Morris Jr, and V. R. Pratt. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6, 2, 323–350. DOI: <https://doi.org/10.1137/0206024>.
- J. Kornerup. 1997a. *Data Structures for Parallel Recursion*. Ph.D. thesis. University of Texas at Austin.
- J. Kornerup. 1997b. Parlists—A generalization of powerlists. In C. Lengauer, M. Griebel, and S. Gorlatch (Eds.), *Proceedings of Euro-Par'97*, Lecture Notes in Computer Science, Vol. 1300. Springer-Verlag, 614–618. DOI: <https://doi.org/10.1007/BFb0002791>.
- J. B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.* 7, 1, 48–50. DOI: <https://doi.org/10.1090/S0002-9939-1956-0078686-7>.
- R. E. Ladner and M. J. Fischer. 1980. Parallel prefix computation. *J. ACM* 27, 4, 831–838. DOI: <https://doi.org/10.1145/322217.322232>.
- W. J. LeVeque. 1956. *Topics in Number Theory*, Vol. 1. Addison-Wesley, Reading, MA.
- S. Marlow (Ed.). 2010. Haskell 2010 Language Report. <https://www.haskell.org/onlinereport/haskell2010/>.
- K. L. McMillan. 2005. Applications of Craig interpolants in model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Vol. 3440. Springer, 1–12. DOI: https://doi.org/10.1007/978-3-540-31980-1_1.
- E. Mendelson. 1987. *Introduction to Mathematical Logic* (3rd. ed.). Wadsworth & Brooks.
- J. Misra. 1977. Prospects and limitations of automatic assertion generation for loop programs. *SIAM J. Comput.* 6, 4, 718–729. DOI: <https://doi.org/10.1137/0206052>.
- J. Misra. 1978. A technique of algorithm construction on sequences. *IEEE Trans. Softw. Eng.* SE-4, 1, 65–69. DOI: <https://doi.org/10.1109/TSE.1978.231467>.
- J. Misra. 1979. Space-time trade off in implementing certain set operations. *Inf. Process. Lett.* 8, 2, 81–85. DOI: [https://doi.org/10.1016/0020-0190\(79\)90148-0](https://doi.org/10.1016/0020-0190(79)90148-0).
- J. Misra. 1981. An exercise in program explanation. *ACM Trans. Program. Lang. Syst.* 3, 1, 104–109. DOI: <https://doi.org/10.1145/357121.357128>.
- J. Misra. 1994. Powerlist: A structure for parallel recursion. *ACM Trans. Program. Lang. Syst.* 16, 6, 1737–1767. DOI: <https://doi.org/10.1145/197320.197356>.

- J. Misra. 2000. Generating-functions of interconnection networks. In J. Davies, B. Roscoe, and J. Woodcock (Eds.), *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare, Cornerstones of Computing*. Macmillan Education UK.
- J. Misra. September. 2021. A proof of Fermat’s little theorem. <http://www.cs.utexas.edu/users/misra/Fermat.pdf>.
- J. Misra and D. Gries. 1982. Finding repeated elements. *Sci. Comput. Program* 2, 2, 143–152. DOI: [https://doi.org/10.1016/0167-6423\(82\)90012-0](https://doi.org/10.1016/0167-6423(82)90012-0).
- C. Morgan. 1990. *Programming from Specifications*. Prentice-Hall, Inc.
- J. B. Orlin. 2013. Max flows in $\mathcal{O}(n \cdot m)$ time, or better. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*. 765–774.
- S. Owicki and D. Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Inform.* 6, 1, 319–340. DOI: <https://doi.org/10.1007/BF00268134>.
- S. Pettie and V. Ramachandran. 2002. An optimal minimum spanning tree algorithm. *J. ACM* 49, 1, 16–34. DOI: <https://doi.org/10.1145/505241.505243>.
- B. Pope. 2001. A tour of the Haskell Prelude. <https://cs.fit.edu/~ryan/cse4250/tourofprelude.html>. Unpublished manuscript.
- E. Price. 2021. Sparse recovery. In T. Roughgarden (Ed.), *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 140–164. DOI: <https://doi.org/10.1017/9781108637435.010>.
- R. C. Prim. 1957. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.* 36, 6, 1389–1401. DOI: <https://doi.org/10.1002/j.1538-7305.1957.tb01515.x>.
- J. Queille and J. Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*. Lecture Notes in Computer Science, Vol. 137. Springer, 337–351. DOI: https://doi.org/10.1007/3-540-11494-7_22.
- J. A. Robinson. 1965. A machine-oriented logic based on the resolution principle. *J. ACM* 12, 1, 23–41. DOI: <https://doi.org/10.1145/321250.321253>.
- C. Rocha and J. Meseguer. 2008. Theorem proving modulo based on boolean equational procedures. In *Proc. ReMiCS 2008*. Lecture Notes in Computer Science, Vol. 4988. Springer, 337–351. DOI: https://doi.org/10.1007/978-3-540-78913-0_25.
- T. Roughgarden and G. Valiant. 2015. CS168: The modern algorithmic toolbox, lecture 2: Approximate heavy hitters and the count-min sketch. <https://web.stanford.edu/class/cs168/l12.pdf>.
- A. Schönhage and V. Strassen. 1971. Schnelle multiplikation großer zahlen. *Computing* 7, 3, 281–292. DOI: <https://doi.org/10.1007/BF02242355>.
- M. Sharir. 1981. A strong-connectivity algorithm and its applications to data flow analysis. *Comput. Math. Appl.* 7, 1, 67–72. DOI: [https://doi.org/10.1016/0898-1221\(81\)90008-0](https://doi.org/10.1016/0898-1221(81)90008-0).
- M. Sollin. 1965. La tracé de canalisation. *Programming, Games, and Transportation Networks*. Wiley.

- V. Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13, 4, 354–356. DOI: <https://doi.org/10.1007/BF02165411>.
- R. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2, 146–160. DOI: <https://doi.org/10.1137/0201010>.
- R. E. Tarjan and J. van Leeuwen. 1984. Worst-case analysis of set union algorithms. *J. ACM* 31, 2, 245–281. DOI: <https://doi.org/10.1145/62.2160>.
- A. Tarski. 1955. A lattice-theoretical fixpoint theorem and its application. *Pac. J. Math.* 5, 2, 285–309. DOI: <https://doi.org/10.2140/pjm.1955.5.285>.
- J. van Heijenoort. 2002. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Source Books in the History of the Sciences. Harvard University Press.
- S. Warshall. 1962. A theorem on boolean matrices. *J. ACM* 9, 1, 11–12. <https://doi.org/10.1145/321105.321107>.
- J. W. J. Williams. 1964. Algorithm 232: Heapsort. *Comm. ACM* 7, 347–348.

Author's Biography

Jayadev Misra



Jayadev Misra is the Schlumberger Centennial Chair Emeritus in computer science and a University Distinguished Teaching Professor Emeritus at the University of Texas at Austin. Professionally he is known for his contributions to the formal aspects of concurrent programming and for jointly spearheading, with Sir Tony Hoare, the project on Verified Software Initiative (VSI).

Jayadev Misra in collaboration with K. Mani Chandy has made a number of important contributions in the area of concurrent computing, among them a programming notation and a logic, called UNITY, to describe concurrent computations, a conservative algorithm for distributed discrete-event simulation, a number of fundamental algorithms for resource allocation (the drinking philosopher's problem, deadlock detection, distributed graph algorithms), and a theory of knowledge transmission in distributed systems. In collaboration with David Gries, Jayadev proposed the first algorithm for the heavy-hitters problem. He has also proposed a set of axioms for concurrent memory access that underlie the theory of linearizability. His most recent research project, called Orc, attempts to develop an algebra of concurrent computing that will help integrate different pieces of software for concurrent execution.

Jayadev is a member of the National Academy of Engineering. He has been awarded: the Harry H. Goode Memorial Award, IEEE, 2017, jointly with K. Mani Chandy; Doctor Honoris Causa, from École normale supérieure Paris-Saclay, Cachan, France, 2010 and a Guggenheim Fellowship, 1988. He is an ACM Fellow (1995), an IEEE Fellow (1992), and a Distinguished alumnus of IIT Kanpur, India.

Index

- 15-puzzle, [169–171](#)
- 2-CNF Boolean expression to graph,
reduction of, [314–315](#)
- 2-satisfiability, [314](#)
- 3-CNF, [314](#)

- Aaronson, Scott, [xx](#)
- Achatz, K., [500](#)
- Ackermann function, [120–121](#), [391](#), [406](#)
- Acyclic directed graph, topological
order in, [303–304](#)
- Acyclic graph, [275–277](#)
- Adams, W., [486](#), [500](#)
- Adjacency list in graph, [270](#)
- All-pair shortest path algorithm, [321](#),
[323](#)
- Alphabetic order. *See* Lexicographic
order
- Alternating path, [288](#)
- Amortized cost, [177](#), [310–311](#)
- Antecedents, [55](#), [152](#)
- Antisymmetric relation, [28](#)
- Arc, [266](#)
- Arithmetic and geometric mean, [102](#),
[207](#)
 - am–gm inequality, [102](#)
 - Dijkstra’s proof of am–gm
inequality, [207](#)
 - proof, [102–104](#)
 - simple proof using
reformulation, [104](#)
- Array
 - notation, [9](#)
 - in powerlists, [492](#)
 - proof rule for assignment,
[154–155](#)
 - in quantified expression, [51](#)
 - segment, [9](#)
- Aspvall, B., [314](#)
- Assertion, [146](#)
- Assignment
 - to array items, [154](#)
 - command, [153–154](#)
 - multiple, [154](#)
- Associativity, [13](#)
 - omitting parentheses with, [42](#)
 - operator, [39–40](#)
 - in quantified expression, [53](#)
- Asymmetric relation, [28](#)
- Asymptotic growth, [176](#), [178–179](#)
- Augmented graph, [302–304](#)
- Auxiliary variables, [147–148](#)
- Axioms, [55](#)
 - of assignment, [154](#)

- Backward substitution rule, [154–155](#)
- Bag, [8](#)
- Balanced binary search tree (bbst),
[448](#)

- Basu, S. K., 165
- Batcher sorting schemes, 479–480
 - batcher odd-even mergesort, 188–189, 480
- Batcher, K., 176, 188, 189, 479
- bbst. *See* Balanced binary search tree (bbst)
- BDD. *See* Binary Decision Diagram (BDD)
- Beckenbach, E., 104
- Bellman, R., 104, 335, 436
- Bellman–Ford algorithm, 335, 338
- Bentley, J. L., 208, 214
- Bentley–McIlroy partition in quicksort, 208, 214
- Better multiplication program, 162, 166–167, 171, 175
- Bézout’s identity, 87, 100, 105, 196
- Bifix
 - characterization, 226–227
 - core, 225–226
- Bijective function, 19–23, 64, 66
- Bijective one-way function, 30
- Binary Decision Diagram (BDD), 275
- Binary expansion, number of
 - non-zero bits in, 407–408
- Binary greatest common divisor, 390
- Binary operators, 13
- Binary relations, 26, 32
 - important attributes of, 28–29
- Binary search, 183, 203–205, 207
- Binary search tree (bst), 425, 448
- Binary tree, 10, 124–125, 274, 424–425
- Binding rules in Haskell, 384–385
- Bipartite graph, 277–278, 281–282, 288, 349
- Bird, R. S., 257
- Bitonic sort, 483–485
- Bloom filter, 31
- Bloom, B. H., 31
- Blum, M., 189
- Bool data type in Haskell, 380
- Boole, George, 40
- Boolean algebra, 40, 471
 - as commutative ring, 43
- Boolean connectives, 37
- Boolean expression, 37–40, 275, 314–315
- Boolean operators, 37, 44, 380
 - functional completeness of, 43–44
 - informal meanings of, 38
- Borůvka, 339, 348
- Borůvka’s Algorithm, 348
- Bottom-up solution procedures, 432–433
 - bottom-up preferable to top-down, 433–435
 - encoding bottom-up solutions in Haskell, 435
- Bound variables in predicate calculus, 50–51
- Boyer, R. S., 223, 255
- Breadth-first traversal algorithm, 292–296
- British Museum algorithm, 141
- Bryant, R. E., 275
- bst. *See* Binary search tree (bst)
- Burstall, R. M., 122
- CAD. *See* Computer-aided design (CAD)
- Cantor, G., 64–65
- Cantor’s diagonalization, 64–66, 97
- Cantor–Schröder–Bernstein Theorem, 20, 65, 70
- Capacity in flow networks, 349–352, 354

- Cardinality of finite set, 7
- Carry absorbing, 491
- Carry generating, 491
- Carry propagating, 491
- Carry-lookahead adder, 485, 490–492
- Cartesian product, 14–15, 26, 323
- Chain in a total order, 32
- Chameleons, 143–144
 - problem revisited, 173
- Chandy, K. M., xx, 365
- Character, 9, 381
- Charikar, M., 254
- Chebyshev, 238, 242
- chr function in Haskell, 381
- Church, A., 496
- City metro system, 321
- Clarke, E., 192
- Clauses, logical, 46–50
 - Clauses in Haskell programming, 386
- CNF. *See* Conjunctive normal form (CNF)
- Coffee can problem, 120, 142–143
 - variation of, 172–173
- Coloring cells in square grid, 169
- Combinational circuit verification, 500
- Command in program, 149
- Commutativity, 13, 39–40, 69, 430–431
- Complement of binary relation, 26
- Complement of set, 12, 14, 42
- Complete induction, 86
- Complete lattice, 70
- Components of graph, 269
- Composite numbers, 233
 - characterization of composite numbers, 237–239
- Composition
 - of functions, 20
 - map function distributes over, 412–413
 - of relations, 26
- Computability, 19
- Computer-aided design (CAD), 275
- Concatenation function for lists, 408–410
- Condensed graph, 272, 313–314, 316
- Conditional command, 149, 156–157
- Conditional equation in Haskell, 386–387
- Conjunction, 45, 47
- Conjunctive normal form (CNF), 45–47, 314
 - reduction of CNF proposition by assigning specific truth values, 46
- Conjuncts, 38, 45, 244, 314–315
- Connected components of graphs, 269, 304
- Consequence rule in program proving, 153
- Constructive existence proofs, 61
- Contract, 146–148
- Contradiction, 56
 - proofs, 59–61
- Contrapositive proofs, 59–61
- Contrapositive rule, 56
- Convexity rule for graphs, 298, 375
- Cook, S., 222
- Cooley, J. M., 475
- Coppersmith, D., 497
- Core computation, 227
 - abstract program for, 228–229
 - representation of prefixes, 229–230
 - running time, 230–231
 - underlying theorem for, 227–228
- Countably infinite set, 66, 82, 112

- Coxeter, H. S. M., 5
- Currying, 400–401
- Cut-set in maximum flow, 354
- Cycles in graphs, 268, 271, 326, 336

- Dahl, O., 140
- Data parallel programs, 459
- Data structuring in Haskell, 382
 - list, 382–383
 - tuple, 382
- Data types in Haskell, 380–382
- Davis, M., 47, 49
- Davis–Putnam–Logemann–Loveland
 - algorithm (DPLL algorithm), 49–50
- De Bruijn graph, 286–287
- De Bruijn sequence, 285–287
- de Bruijn, N. G., 285
- De Morgan’s law, 41, 43–46, 52–53, 471
- Deadlock in concurrent systems, 168–169
- Degree of node, 364
- Dense graph, 268
- Depth-first traversal, 289, 296–303, 311, 373–376
 - additional properties of, 376
 - algorithm, 296
 - depth-first traversal of
 - undirected graph, 303
 - of directed graph, 299–301
 - proof of edge-ancestor lemma, 373–374
 - proof of path-ancestor theorem, 375–376
 - properties of depth-first traversal
 - of directed graph, 301–302
 - properties of preorder and postorder numbering, 297–299
 - of tree, 296–297
 - visiting all nodes of graph, 302–303
- Derived variables, 147–148
- Dershowitz, N., 121
- Dershowitz–Manna order, 121–122
- Dictionary order. *See* Lexicographic order
- Difference of sets, 12, 15
- Dijkstra, E. W., xi, xx, 2–5, 23–24, 42, 63, 65, 102, 140, 142n4, 150, 166–167, 169, 196, 204, 207–208, 257, 281, 285, 328–329, 332, 335, 338, 340, 343, 346–348, 368, 378
- Dijkstra’s proof of am-gm inequality, 207–208
- Dijkstra–Prim Algorithm, 346–348
- Directed acyclic graph, 275
- Directed graph, 266–269. *See also* Undirected graph
 - breadth-first traversal of, 293
 - condensed graph, 313–314
 - depth-first traversal of, 299–301
 - properties of, 301–302
 - strongly connected components
 - of, 311–313
- Disjunction, 45, 47, 56
- Disjunctive normal form (DNF), 45–47, 275
- Disjuncts, 38, 45–46, 314
- Distributivity, 13
 - in flow networks, 354
 - in powerlists, 463
 - in propositional logic, 40, 42–43, 46
- “Divide and Conquer” approach, 183, 187
 - batcher odd-even sort, 188–189

- long multiplication, [187–188](#)
 - median-finding, [189–190](#)
- DNF. *See* Disjunctive normal form (DNF)
- Domain knowledge, [3–4](#)
- Domino tiling, [282–284](#)
- Double Ackermann function, inverse of, [310](#)
- DPLL algorithm. *See* Davis–Putnam–Logemann–Loveland algorithm (DPLL algorithm)
- Duality principle, [52–53](#)
- Dynamic graph, [305](#). *See also* Directed graph
- Dynamic programming, [378](#), [432](#)
 - top–down vs. bottom–up
 - solution procedures, [432–435](#)
- Edge in a graph, [9](#), [265–267](#)
- Edge-ancestor lemma, [301–302](#), [304](#)
 - proof of, [373–375](#)
- Edges during depth-first traversal, identifying different kinds of, [301](#)
- Elementary facts about sorting, [480–482](#)
- Elementary functions over lists in Haskell, [399](#)
- Elementary proof construction, [56](#)
- Embedding problem in hypercube, [499](#)
- Emerson, E., [192](#)
- Encrypted password, [22](#)
- Equivalence classes, [29–31](#), [306](#)
- Equivalence of two notions of well-foundedness, [118](#)
- Equivalence relation, [29](#), [57](#), [305](#), [429](#)
 - bloom filter, [31](#)
 - checking for, [30](#)
- Eratosthenes, [233](#), [234](#), [237](#)
- Eratosthenes, sieve of, [233–234](#), [239](#)
- Erdős, P., [24](#)
- Escher, M. C., [83](#)
- Euclid, [61](#), [196](#), [389](#)
- Euclid’s proof of infinity of primes, [61](#)
- Euler, [89](#), [278](#), [279](#)
- Euler path and cycle, [278–280](#)
 - in directed graph, [280](#)
 - Euler cycle in de Bruijn graph, [286–287](#)
- Euler-Binet formula, [89](#)
- Event_queue, [334–335](#)
- EWDs, [3](#)
- Exclusive-or, [43](#)
- Existentially quantified proof, [56](#)
- Expansion in solving recurrence relations, [185–186](#)
- Extension heuristic for function generalization over lists, [417–418](#)
- Farach-Colton, M., [254](#)
- Fast Fourier Transform (FFT), [475](#)
 - computing product of polynomials, [479](#)
 - derivation of, [476–478](#)
 - Inverse Fourier transform, [478–479](#)
- Feijen, W.H.J., [57](#)
- Fermat, [26](#)
- Fermat’s little theorem, [284–285](#)
- FFT. *See* Fast Fourier Transform (FFT)
- Fibonacci numbers, [88](#), [406–407](#)
 - closed form for, [89](#)
 - computation in Haskell, [388](#)
- Fibonacci of Pisa, [88](#)ⁿ¹
- Finite 2-player game, winning strategy in, [113–115](#)

- Finite state transducer, 404
 - simulate finite state transducer by program, 405
- Fischer, M. J., 489
- Fitting, M., 37
- flatten function for nested lists, 399, 402
- Float data type in Haskell, 380
- Flow in capacitated network, 349
 - augmenting path, 1, 350
 - value, 350
- Floyd, R. W., 140, 215, 220, 259, 321
- fold* function, 474
- foldr function
 - foldr function in Haskell, 401–403
 - foldr* function for powerlists, 474
- for command, 150
- Ford Jr, L. R., 1, 335, 349, 351, 353
- Ford–Fulkerson algorithm for
 - maximum flow, 353
- Forest, 10, 282, 302
- Formal proofs, 54
 - strategies, 55–56
 - style of writing proofs, 56–59
- Formal treatment of partial
 - correctness, 145
 - auxiliary and derived variables, 147–148
 - hoare-triple or contract, 146–147
 - specification of partial correctness, 145–146
- Fourier transform. *See* Fast Fourier Transform
- Fredman, M., 310
- Free tree, 274–275
- Free variables in predicate calculus, 50
- Frege, Gottlob, 11, 63
- Fulkerson, D., 1, 349, 351, 353
- Full binary tree, 133, 221, 425–427
- Function call, 152, 404
- Function definition in Haskell, 378, 383
 - binding rules, 384–385
 - conditional equation, 386–387
 - multiple equations in, 385–386
 - sequence of, 390
 - using where clause, 387
 - writing, 385
- Function generalization, 416
 - enumerating permutations, 421–422
 - extension heuristic for function generalization over lists, 417–418
 - list reversal, 420–421
 - maximum segment sum, 418–419
 - prefix sum, 419–420
- Function map, 403–404
 - distributes over composition, 412–413
- Functional programming, 378, 384, 420, 435, 459–460
- Functions, 10, 17, 388, 399
 - arity, 18–19
 - associative function composition, 412
 - basic concepts, 17–20
 - Cantor-Schröder-Bernstein Theorem, 20
 - composition, 20–21
 - count function, 407
 - definition and computability, 19
 - families, 176–177
 - filter function, 411
 - hierarchy, 179–180
 - higher-order function, 23

- inverse, [21](#)
- monotonic function, [22–23](#)
- one-way function, [22](#)
- order, [181–182](#)
- pigeonhole principle, [23–25](#)
- specification, [161](#)
- taxonomy, [19–20](#)
- Fundamental cycle, [340](#)
- Fundamental theorem for MST,
 - [340–343](#)
 - abstract algorithm for MST, [343–344](#)
 - independence among safe edges, [341–343](#)
 - safe edge, [341](#)
- Fundamental theorem of arithmetic.
 - See* Unique prime factorization theorem
- Fürer, M., [188](#)
- Gale, D., [245](#)
- Gardner, M., [65](#)
- Gates in circuits, [44](#)
- gcd. *See* Greatest common divisor (gcd)
- General recursion, [391](#)
- General tree, [427–428](#)
- Goldberg, A. V., [349](#), [356](#)
- Goldberg–Tarjan max-flow algorithm, [356–364](#)
 - complete algorithm, [358](#)
 - correctness, [359](#)
 - details of algorithm, [357](#)
 - initialization, [357](#)
 - invariant, [359–361](#)
 - effect of non-saturating push, [363](#)
 - preconditions of push and *relabel*, [361](#)
 - preflow* max flow at termination, [361](#)
 - push operation, [357](#)
 - effect of *relabel*, [362](#)
 - relabel* operation, [358](#)
 - effect of saturating push, [362](#)
 - termination, [361–362](#)
 - timing analysis, [362](#)
- Golden ratio, [89n2](#)
- Gouda, Mohamed, [xx](#)
- Grace, Karen, [xx](#)
- Graph manipulation, [271–273](#)
 - adding/removing nodes/edges, [271–272](#)
 - join disjoint cycles, [273](#)
 - join non-disjoint cycles, [273](#)
 - join or replace paths, [273](#)
 - merging nodes, [272](#)
- Graph representation, [270–271](#)
- Graph traversal algorithms, [292–293](#)
- Grasp, [178](#)
- Gray code, [448](#), [471](#)
 - function, [471](#)
 - in multi-dimensional powerlist, [489](#)
- Greatest common divisor (gcd), [36](#), [87](#), [165](#), [389–390](#)
- Greatest lower bound (glb), [35–37](#), [68](#), [70](#)
- Grid line, [281](#)
- Grid point, [281](#)
- Gries, David, [xx](#), [3](#), [37](#), [52](#), [90](#), [142](#), [172n11](#), [205](#), [250](#), [253](#)
- Gupta, A., [418](#)
- Hadamard matrix, [108–110](#)
- Half-step puzzle, [94–95](#), [99](#)
- Hall condition (HC), [288](#)
- Hall’s marriage theorem, [287–289](#)

- Hall, P., 287
- Hamiltonian path and cycle, 280–281
 - Hamiltonian cycle in domino tiling, 283–284
- Hamming distance, 108
- Handshake problem, 24
- Hard problems, 178
- Hardy, 102
- Harmonic numbers, 89–90
- HC. *See* Hall condition (HC)
- Heapsort, 215
 - extendHeap*, 218–220
 - heap data structure, 216–218
 - heapify*, 216, 220–221
 - main program, 216
 - running time, 220–221
- Heavy-hitters, 250
 - abstract algorithm and refinement, 251–254
 - majority element of bag, 255–256
 - one-pass algorithm for approximate heavy-hitters, 254–255
- Hegner, E. C. R., xx, 3, 150, 173, 204
- Helper function, 415, 417, 480, 483
 - enumerating permutations, 415–416
- Hendrix, J., 500
- Heule, M. J. H., 45
- Heuristics for postulating invariant, 164–167
- Hierarchical refinement, 2
- Higher order functions, 23, 400
 - currying, 400–401
 - foldr* function, 401–403
 - function composition is associative, 412
 - function map, 403–404
 - function map distributes over composition, 412–413
 - proof about *filter*, 411–412
 - reasoning about, 410
- Higher-dimensional constructors for powerlists, 506–508
- Hinkis, A., 20
- Hoare, C. A. R., 140, 193, 208
- Hoare, T., xx, 146, 192, 215, 378n1
- Hoare-triple, 146–147
- Horner's rule for sequential polynomial evaluation, 414
- Hudak, P., 378
- Hypercubes, 460
 - embedding in, 498–500
- Hypercubic algorithms, 459
- Hypotheses, 55, 64, 152
- Identity
 - function, 20
 - relation, 26
- In-degree of node, 267
- In-span cost, 311
- Induction, 12, 32, 83, 145, 378
 - advanced examples, 101–115
 - applying base step, 93–94
 - examples about games and puzzles, 90–93
 - examples from arithmetic, 87–90
 - examples of proof, 87–93
 - exercises, 125–137
 - generalization, 95
 - hypothesis, 85
 - methodologies for applying, 93–101
 - misapplication of induction, 97–101
 - motivating induction principle, 84–85

- Noetherian induction, 115–122
- overuse, 99–100
- problem reformulation,
 - specialization, 95–96
- proof by contradiction versus
 - proof by induction, 97
- strengthening of induction
 - hypothesis, 94–95
- strong induction principle over
 - natural numbers, 86–87
- structural induction, 122–125
- weak induction principle over
 - natural numbers, 85–86
- Infinite structures, proofs over,
 - 100–101
- Infinite tree, 100–101, 113
- Infix operators, 53, 379–380, 402
- Informal writing, 18
- Injective function, 19–20, 23, 64–66
- Input specification, 145–146
- Int data type in Haskell, 380
- Integer interval, 9, 36
- Inter-component edge, 341–343,
 - 345–346
- Interpolation
 - in solving recurrence relations,
 - 185
 - search, 204
- Intersection of sets, 12–14
- Interval, 9, 36
 - of complete lattice, 71
- Invariant, 2, 141–145, 164
 - heuristics for postulating,
 - 164–167
 - non-termination, 168–171
 - perpetual truth, 167–168
 - strength, 167
- Inverse Fourier transform, 478–479.
 - See also* Fast Fourier Transform (FFT)
- Inverse of a function, 21
- Inversion function on powerlists,
 - 470–471
- Iterated logarithm, 199, 310–311
- Joffrain, Thierry, xx
- Join by rank, optimization heuristic
 - for, 307
- Join disjoint cycles, 273
- Join non-disjoint cycles, 273, 280
- Join paths, 273
- Kapur, D., 500
- Karatsuba's algorithm, 187–188
- Kierkegaard, Søren, 153
- King, V., 349
- Kitchin, D., 209
- Knaster-Tarski theorem, 20, 36, 70–74
- Knuth, D. E., 176, 222, 259
- Knuth-Morris-Pratt string-matching
 - algorithm, 222
 - abstract KMP program, 232
 - concrete KMP program, 232–233
 - core computation, 227–231
 - string-matching problem and
 - outline of algorithm, 222–225
 - underlying theory, 225–227
- König's lemma, 101, 113, 122, 134
- Kornerup, J., 500
- Kosaraju, S. Rao, xx, 311
- Kruskal's algorithm, 343, 345–348
- Kruskal, J. B., 339, 345
- Labeled graph, 269–270
- Ladner, R. E., 489
- Ladner-Fischer algorithm, 489
 - complexity of, 489

- Large graphs, 266
- Lattice, 36–37, 70
- Laws
 - of powerlist, 463
 - of predicate calculus, 51–52
 - of propositional logic, 40–41
 - about sets, 15–16
 - about subsets, 16
- lcs. *See* Longest common subsequence (lcs)
- Leaf, 10, 424–427, 431
- Least upper bound, 35–36
 - properties of least upper bound of partial order, 68–69
- Least-significant digit sort, 33–34
- LeVeque, W. J., 238
- Lexicographic order, 33, 100, 102, 116–120, 172–173, 198, 395, 397
- Li, Xiaozhou (Steve), xx
- Lifschitz, Vladimir, xx, 48n5, 63
- Linear order. *See* Total order
- Linked list representation of graph, 270
- List data structure in Haskell, 382–383
 - reversal, 420–421
- Literals in logical formula, 38
- Littlewood, 102
- Long multiplication, 187–188
- Longest common subsequence (lcs), 439–443, 445, 451, 456
- Longest path in graph, 327–328, 418
- Loop command, 149–150, 157–159, 200
- Loyd, Sam, 169

- Mamata, Misra, xx
- Manna, Z., 121
- mapp* operation, 465
- Marlow, S., 378

- Master Theorem, 186–187, 189–190, 480, 489
- Mathematical objects, 7–8
 - array, matrix, 9
 - character, 9
 - functions and relations, 10
 - integer interval, 9
 - sequence, 8
 - string, 9
 - tree, 9–10
 - tuple, 8
- Matrix, 9, 13
 - chain multiplication, 438–439
 - Hadamard, 108–110
 - problem in proving termination, 174
 - in transitive closure, 319, 323–324
 - transitive closure and matrix multiplication, 318–319
 - transposition, 403, 495–496
- Maude ITP, 501
- Max-flow min-cut theorem, 1, 349, 354–356
 - algebra of flows, 354
 - statement and proof of, 354–356
- Maximum flow, 1–2, 349–364
 - Ford-Fulkerson algorithm for, 352–353
 - Goldberg–Tarjan algorithm for, 356–364
 - max-flow min-cut theorem, 354–356
 - problem specification, 349–351
 - residual network, 351
- Maximum segment sum, 327–328, 418–419
- McCarthy, John, 110
- McCarthy’s 91 function, 110
- McIlroy, M. D., xx, 33n3, 208, 214

- McMillan, K. L., [185](#)
- mdp. *See* Multi-dimensional powerlists (mdp)
- Median-finding, [189–190](#)
- Mendelson, E., [37](#)
- Merge-sort, [176, 181, 184, 400](#)
- Merging nodes, [271–272](#)
- Meseguer, J., [43](#)
- Metric for program termination, [144, 172](#)
- Metro connection directory, [321](#)
 - compiling, [321–323](#)
 - definitions of $\times$ and $+$ for metro directory, [323](#)
 - Warshall’s Algorithm for, [324](#)
- `min` function in Haskell, [379–380](#)
- Minimum cut, [354–356](#)
- Minimum edit distance, [442](#)
 - dynamic programming solution, [443–445](#)
 - program to compute edit distance, [443](#)
 - underlying mathematics, [442–443](#)
- Minimum spanning tree (MST), [305, 339–348](#)
 - abstract algorithm for, [343–344](#)
 - Borůvka’s algorithm, [348](#)
 - Dijkstra–Prim algorithm, [346–348](#)
 - fundamental theorem about minimum spanning trees, [340–344](#)
 - Kruskal’s algorithm, [345–346](#)
 - properties of spanning tree, [340](#)
- Misapplication of induction, [97–101](#)
- Misra, J., [xx, 65, 165, 166, 193, 233, 245, 250, 253, 285, 365, 418, 451, 500](#)
- Model Checking, [192](#)
- Modest programming language, [148–152](#)
- Monotonic function, [22–23, 179](#)
 - Monotonically decreasing function, [22](#)
 - Monotonically increasing function, [22](#)
 - over partially ordered sets, [37](#)
- Monotonic sequence, [23, 134–135](#)
 - Longest increasing subsequence, [451](#)
 - Monotone subsequence, minimum length of, [24–25](#)
- Moore, J. S., [xx, 223, 255](#)
- Morgan, Carroll, [3, 174n13](#)
- Morris Jr, J. H., [222](#)
- MST. *See* Minimum spanning tree (MST)
- Multi-dimensional array, [492](#)
- Multi-dimensional matrix, [459–460](#)
- Multi-dimensional powerlists (mdp), [462, 466, 492](#)
 - algorithms, [495](#)
 - depth, shape, [494–495](#)
 - embedding in hypercubes, [489–500](#)
 - gray code in, [489](#)
 - indices of nodes in, [495](#)
 - matrix transposition, [495–496](#)
 - multi-dimensional constructors, [494](#)
 - outer product, [495](#)
 - proofs over, [495](#)
 - properties of multi-dimensional constructors, [494](#)
 - square matrix multiplication, [497](#)
 - Strassen’s matrix multiplication algorithm, [497–498](#)

- transposition of square matrix, 496–497
- Multiary relation, 26
- Multiple assignment, 149, 151, 154–155
- Multiple equations in Haskell
 - function definition, 385–386
- Multiset, 8
 - ordering, 121–122
- Mutual implication, 55–56, 68, 118, 200, 226, 238, 290, 336
- Mutual recursion, 404–406
- n -ary relation. *See* Multiary relation
- n -dimensional hypercube, 498–499
- n -tuple, 8, 26, 33–34, 117
- Naive comprehension, 11
- Negative edge lengths, shortest paths
 - with, 335–339
- Neural networks, 265
- Neurons, 266
- Nodes in graphs, 9–10, 265–266, 269
- Noether, Emmy, 115 n 4
- Noetherian induction, 115
 - multiset or bag ordering, 121–122
 - well-founded relation, 116–121
- Non-Boolean domains, quantified
 - expressions over, 53–54
- Non-constructive existence proofs, 61
- Non-constructive strategy, 56
- Non-determinism, 150–151
- Non-deterministic assignment, 150–151, 159–160, 173, 204
- Non-negative edge lengths, shortest paths with, 328–332
- Non-standard powerlists, 464
- Non-termination, 143, 168–171, 437
- Non-zero bits in binary expansion, number of, 407–408
- Notational devices, 504
- NP -complete, 178, 281
- Objective function, 435–437
- One-pass algorithm for approximate heavy-hitters, 254–255
- One-to-one function maps. *See* Injective function maps
- One-way function, 22
- Onto function maps. *See* Surjective function maps
- Operators
 - binary, 13
 - boolean, 37, 38
 - commutative ring with, 43
 - functional completeness of, 43–44
 - omitting parentheses with, 42
 - precedence of, 39
 - dual of, 53
 - identity, 53
 - infix, 53
 - powerlist constructors, 461
 - prefix and infix in Haskell, 379
 - right-shift in powerlist, 486
 - scalar in powerlist, 464
 - semi-distributes over multiplication, 179
 - semiring, 324
 - on sets, 12
 - transitive, 29
 - unit elements of, 53
- Optimal solution, 436–437
- Optimality principle, 435–438
- Optimization heuristics, union-find algorithm, 307–309
 - for find, 308
 - for union, 307
- ord function in Haskell, 381, 385, 395
- Order of functions, 178

- function hierarchy, 179–180
- function order, 181–182
- Order relations, 32, 116
 - partial order, 34–37
 - total order, 32–34
- Ordering of subproblems, 98–99
- Orlin, J. B., 349
- Out-degree of graph node, 267, 274, 280, 287
- Output specification, 145–146
- Owicki, S., 172n11
- Pairing points with non-intersecting lines, 60, 173–174
- Panini, 285
- Parallel computations, role of
 - recursion and induction in, 459
- Parallel polynomial evaluation, 414
- Parallel program, 459
- Parallel recursion, 415, 459
 - advanced examples using
 - powerlists, 474–492
 - hypercube embedding
 - algorithm, 508–510
 - inductive measures on
 - powerlists, 466–467
 - laws about higher-dimensional
 - constructors, 506–508
 - multi-dimensional powerlist, 492–500
 - parallelism and recursion, 459–460
 - permutation functions, 467–472
 - pointwise function application, 464–466
 - powerlist, 460–464, 500–501
 - proofs about powerlists, 466
- Parallelism and recursion, 459–460
- Partial correctness, 145, 152
- Partial order, 32, 34, 434
 - interval, 36
 - lattice, 36–37
 - least upper bound, greatest lower bound, 35–36
 - monotonic function over
 - partially ordered sets, 37
 - topological order, 35
- Path-ancestor theorem, 301–302
 - proof of, 375–376
- Paths, 10, 32, 99, 268, 317, 324, 329, 436
 - compression, 308
- Paths and cycles, 268
- Pattern matching in Haskell, 397
 - examples of, 399
 - in powerlists, 478, 496
 - rules, 398
 - wildcard, 398
- Patterns in computation, 377
- Peano axioms, 459
- Pebble movement game, 92–93
- Permutation, 30, 210, 472
 - correctness of, 472–473
 - enumeration, 415–416, 421–422
 - examples of, 468
 - functions, 467–468
 - gray code, 471
 - inversion, 470–471
 - permutations distribute over
 - scalar functions, 472
 - permute index, 470
 - proof of *revp*, 473–474
 - rotate, 468
 - swap, 468–469
 - using transposition, 101–102
- Permute index function, 470
- Perpetual truth, 167–168

- Personal disagreement with Haskell's
 - choice of symbols, 387
- PERT chart, 270
- Pettie, S., 340
- Photons, 333–334
- Pidgeon, Sean, xx
- Pigeonhole principle, 23, 260
 - minimum length of monotone
 - subsequence, 24–25
- Plaxton, Greg, xx, 320n4
- Pointwise function application,
 - 464–465
 - examples of scalar function
 - application, 465–466
 - powerlist of indices, 466
- Polar coordinates, 21, 396
- Pólya, 102
- Polymorphism, 393–394
- Polynomial, 5
 - computing product of, 479
 - evaluation
 - parallel, 474–475
 - sequential, 414
 - function, 474–475
- Pope, B., 378
- Positive edge in maximum flow, 350,
 - 357, 360
- Positive path in maximum flow,
 - 350–353, 355, 359–361
- Postorder numbering, properties of,
 - 297–299
- Powerlists, 459–460
 - advanced examples using, 474
 - batcher sorting schemes, 479–485
 - constructors, 461–462, 492
 - FFT, 475–479
 - fold*, 474
 - inductive measures on, 466–467
 - laws, 463–464
 - non-standard powerlists, 464
 - polynomial, 474–475
 - prefix sum, 485–492
 - simple function definitions over,
 - 462–463
- Powerset, 12, 64–66
- Pratt, V. R., 222
- Precedences of operators in
 - propositional logic, 39
- Predicate calculus, 45, 50
 - duality principle, 52–53
 - free and bound variables, 50
 - laws, 51–52
 - quantified expressions over
 - non-Boolean domains, 53–54
 - syntax of quantified expressions,
 - 50–51
- Prefix operators in Haskell, 379–380
- Prefix sum, 419–420, 485–486
 - carry-lookahead adder, 490–492
 - complexity of Ladner–Fischer
 - algorithm, 489
 - properties of, 487–488
 - remarks on prefix sum
 - description, 490
 - simple algorithm for, 488–489
 - specification, 486–487
- Preflow in maximum flow algorithm,
 - 356–359
 - max flow at termination, 361
- Premises, 55
- Preorder number, 296–297, 299, 304
 - Preorder numbering, properties
 - of, 297–299
- Price, E., 250
- Prim, R. C., 340, 346
- Primitive recursion, 391
- Problem reformulation in induction,
 - 95–96, 107, 109

- Procedure call, [151](#), [160–161](#)
- Procedure implementation, [161](#)
- Product puzzle, sum and product puzzle, [61–63](#)
- Program layout in Haskell, [387–388](#)
- Programming in Haskell, [378–379](#)
 - basic data types, [380–382](#)
 - data structuring, [382–383](#)
 - elementary functions over lists, [399](#)
 - examples of pattern matching over lists, [399](#)
 - function definition, [383–387](#)
 - higher order functions, [400–404](#)
 - list concatenation and flattening, [399–400](#)
 - merge sort, [400](#)
 - mutual recursion, [404–406](#)
 - pattern matching, [397–398](#)
 - polymorphism, [393–394](#)
 - program layout, [387–388](#)
 - recursively defined functions, [388–391](#)
 - type, [391](#)
 - type classes, [394](#)
 - type expression, [392–393](#)
 - user-defined types, [395–397](#)
- Programs, imperative
 - analysis, [140](#)
 - annotation, [161–164](#)
 - state, [146](#)
 - verification problem, [139–141](#)
- Proof by contradiction, [56](#), [97](#)
- Proof by contradiction vs. proof by induction, [87](#), [90](#), [94](#), [97](#), [115](#)
- Proof construction, [55](#), [59](#)
 - constructive and non-constructive existence proofs, [61](#)
 - diagonalization (Cantor), [64–66](#)
 - Knaster-Tarski theorem, [70–74](#)
 - proofs by contradiction, contrapositive, [59–61](#)
 - properties of least upper bound of partial order, [68–69](#)
 - Russell's paradox, [63–64](#)
 - saddle point, [66–67](#)
 - sum and product puzzle, [61–63](#)
- Proof of reachability program, [371–373](#)
- Proof rules, [152–155](#)
- Propositional logic, [37](#)
 - additional aspects, [42–44](#)
 - basic concepts, [37–40](#)
 - laws of, [40–41](#)
 - satisfiability, validity, [44–50](#)
- Putnam, H., [47](#)
- Puzzles
 - 15-Puzzle, [169–171](#)
 - Chameleons, [143](#), [164](#), [168](#), [173](#), [201–202](#)
 - The coffee can problem, [119](#), [120](#), [142–145](#), [164](#), [172](#)
 - Coloring grid points, [281–282](#)
 - The Half-step puzzle, [94](#), [99](#)
 - A pebble movement game, [92–93](#)
 - Seven Bridges of Königsberg, [278](#), [280](#)
 - Sum and Product, [61–63](#)
 - Termination of a game, [106](#)
 - Trimino tiling, [90–91](#)
- Quantified expressions, [50](#), [53–54](#)
- Queille, J., [192](#)
- quickMlt function, [389](#)
- Quicksort, [164](#), [181](#), [208](#)
 - cost, [212–215](#)
 - partition* procedure, [212](#)
 - procedure, [211–212](#)

- specification, 209–210
- Rabbit (Coxeter's), 4–5
- Ramachandran, V., 340
- Rank in shortest path, 329
- Rank in union–find algorithm, 308, 311
- Reachability in graphs, 269, 289–292
 - abstract program for, 291
 - correctness proof, 291–292
 - definition and properties of, 289–291
 - proof of reachability program, 371–373
- Real interval, 36, 66
- Real-time concurrent program for shortest path, 333
- Reasoning about programs
 - formal treatment of partial correctness, 145–148
 - formal treatment of termination, 171–174
 - fundamental ideas, 141–145
 - invariant, 164–171
 - modest programming language, 148–152
 - order of functions, 178–182
 - program verification problem, 139–141
 - proof rules, 152–164
 - proving programs in practice, 190–193
 - reasoning about performance of algorithms, 175–178
 - recurrence relations, 182–190
 - termination in chameleons problem, 201–202
- Rectangular coordinates, 21
- Recurrence equation. *See* Recurrence relations
- Recurrence relations, 182
 - Divide and Conquer, 187–190
 - interpolation, 185
 - Master Theorem, 186–187
 - merge-sort, 184
 - searching sorted and unsorted lists, 183–184
 - solving, 185
- Recursion, 2–3, 83, 377, 421, 429, 459, 477
- Recursive data types in Haskell, 424
 - tree isomorphism, 428–432
 - tree structures, 424–428
- Recursive problem formulations, alternative, 414–415
- Recursive programming
 - alternative recursive problem formulations, 414–415
 - dynamic programming, 432–445
 - Fibonacci numbers, 406–407
 - function generalization, 416
 - Haskell programming notation, 379–406
 - helper function, 415
 - improving a recursive solution, 422
 - number of non-zero bits in binary expansion, 407–408
 - reasoning about higher order functions, 410–413
 - reasoning about recursive programs, 406–413
 - recursive data types, 424–432
 - recursive programming methodology, 413
 - refining recursive solution, 422–424

- reversal and concatenation
 - function for lists, 408–410
 - towers of Hanoi, 453–456
- Recursive structures, defining, 122–124
- Recursively defined functions in
 - Haskell, 388, 391
 - computing Fibonacci numbers, 388
 - computing pairs of Fibonacci Numbers, 388
 - greatest common divisor, 389–390
 - length of list, 388
 - multiplication of two numbers, 389
 - primitive vs. general recursion, 391
 - writing sequence of function definitions, 390
- Reduction of 2-CNF Boolean
 - expression to graph, 314–315
- Reed–Muller code, 108
- Reflexive closure, 31–32, 291
- Reflexive relation, 28
- Relations, 10, 25
 - basic concepts, 25–26
 - closure, 31–32
 - equivalence relation, 29–31
 - important attributes of binary relations, 28–29
 - relational databases, 26–28
- Replace paths in graphs, 273
- Representative, unique in equivalence class, 30
- Residual network in maximum flow, 351, 353, 356
- Resolution principle, 46–49
 - Resolution rule, 47–48
 - Resolvent, 47, 76
- Resource usage, 175–176
- Reversal function for lists, 408–410
- Rewrite Rule Laboratory (RRL), 500
- Rich, Elaine, xx
- Right-shift operator in powerlist, 486
- Robinson, J. A., 47
- Rocha, C., 43
- Root in a graph, 9, 114, 274, 289, 298, 302, 306, 325, 329, 333
- Rooted tree, 9–10, 274–275
 - breadth-first traversal of, 295
 - depth-first traversal of, 296
- rotate* function in powerlist, 468
- Roughgarden, T., 250
- RRL. *See* Rewrite Rule Laboratory (RRL)
- Russell’s paradox, 63–64
- Russell, Bertrand, 11, 63, 64
- Rutger Dijkstra, 106n3
- Saddle point, 66–67
- Saddleback search, 205–207
- Safe edge, 341, 345, 348
 - independence among safe edges, 341–343
- Saks, M., 310
- SAT-solving problem, 45, 178
 - SAT-solvers, 178
- Satisfiability, 44–50, 62–63
 - 2-Satisfiability algorithm, 314–317
- Scalar binary operator in powerlist, 464, 472
- Scalar functions in powerlists, 464–465, 472, 503
 - application, 465–466
- Scalars in powerlist, 460–461
- Schneider, F. B., 37, 52
- Scholten, C. S., 42, 142n4
- Schönhage A., 188, 479

- Schulte, W., 500
- SDR. *See* System of distinct representatives (SDR)
- Security class, 36–37
- Segments of sequence, 8
- Selection sort, 215
- Self loop in graph, 293, 302
- Semiring, 324
 - transitive closure of, 324
- Sequence, 8
 - comprehension, 12
 - comprehension using total order, 33
 - operations, 15
- Sequencing command, 149, 155–156
- Set, 7
 - alternative to definition by comprehension, 40
 - basic concepts, 7–8
 - comprehension, 10–12
 - mathematical objects as set, 8–10
 - membership, 8
 - operations on, 12–15
 - properties of set operations, 15–17
 - relationship of propositional laws with set algebra, 42
 - subset, 12
- Seven Bridges of Königsberg, 278
- Shade, Bernadette, xx
- Shapley, L. S., 245
- Sharir, M., 311
- Shortest paths, 99, 317, 321
 - cost analysis, Bellman-Ford algorithm, 337–338
 - development of Dijkstra's algorithm, 329–332
 - edge relaxation, Bellman-Ford algorithm, 338–339
 - equation, 326–327
 - implementation, Bellman-Ford algorithm, 337
 - mathematical basis of Dijkstra's algorithm, 328–329
 - negative cycles, 336–337
 - with negative edge lengths, 335–336
 - with non-negative edge lengths, 328
 - real-time concurrent program for, 333
 - underlying equations, Bellman-Ford algorithm, 336
- Sieve algorithm for prime numbers, 233
 - characterization of composite numbers, 237–239
 - correctness, 234–237
 - refinement of sieve program, 239–245
 - sieve of Eratosthenes, 234
- Sifakis, J., 192
- Simple assignment, 149
- Simple commands, 149, 153–155, 162
- Simple path in graphs, 268, 273, 286, 318–319, 326, 336
- Simulation-based shortest path algorithm, 332–333
 - real-time concurrent program for shortest path, 333
 - simulation-based implementation, 334–335
- Single source shortest path, 325–339
 - formal description of problem, 325–326
 - shortest paths with negative edge lengths, 335–339

- shortest paths with non-negative edge lengths, 328–332
 - shortest-path equation, 326–327
 - simulation-based shortest path algorithm, 332–335
 - single source shortest path in acyclic graph, 327–328
- skip* command, 153
- Sollin, M., 348
- Sorted list searching, 183–184
 - Binary search, 203
- Soundness of Resolution principle, 47–48
- Source, directed graph, 267, 328, 349
- Spanning tree, 339
 - properties of, 340
- Sparse graph, 268
- Specialization in induction, 95–96
- Specific graph structures, 274
- Specification of partial correctness, 145–146
- Square matrix, powerlist
 - multiplication, 497
 - transposition of, 496–497
- Stable marriage. *See* Stable matching
- Stable matching, 245
 - correctness, 247–248
 - formal description of problem and algorithm, 246–247
 - refinement of algorithm, 249–250
- Static typing in Haskell, 391
- Strassen's matrix multiplication
 - algorithm, 497–498
- Strassen, V., 188, 479, 497
- Strengthening predicates, 42, 94–95
- Strict order in set, 32–33
- String, 9, 381
 - removing unnecessary white spaces from, 404
- Strong induction principle over natural numbers, 86–87
- Strongly connected component, 269, 272, 311–317
- Structural induction, 116, 122
 - principle, 124–125
 - recursive structures, 122–124
- Style of writing proofs, 56–59
- Subramaniam, M., 500
- Subsequence, 8, 24–25, 116, 439, 442–443
- Subset, 12, 16
- Substring, 9, 25, 28, 126, 285
- Subtree, 9–10, 28, 115, 122, 125, 217–218, 221, 274, 276, 296–297, 303–304, 307, 313, 406, 424, 427
- Successor sets of graph nodes, 270
- Sum and product puzzle, 61–63
- Sum of cubes of successive integers, 87
- Swap function, powerlist, 468–469
- Symmetric closure of binary relation, 31–32, 76
- Symmetric difference of sets, 15
- Symmetric relation, 28–29, 269
- Synchronous parallel programs, 459–460
- System of distinct representatives (SDR), 287
- Tail-recursive function, 399
- Tarjan, R. E., 292, 310, 349, 356
- Tarski, A., 70
- Tentative path lengths, Dijkstra's algorithm, 329
- Termination, 144–145
 - chameleons problem, 173
 - formal treatment of, 171

- of game, 106–108
 - pairing points with
 - non-intersecting lines, 173–174
 - problem on matrices, 174
 - variation of coffee can problem, 172–173
- Testing, program, 139–140, 192, 203
- Theorem proving, 45
- tie* constructor, powerlist, 461
- Tiling, domino, 283
- Top-down formulation of dynamic programming solution, 436
- Top-down solution, 432, 439
 - bottom-up preferable to top-down, 433–435
 - encoding bottom-up solutions in Haskell, 435
 - top-down solution procedures, 432–433
- Topological order, 35
 - in acyclic directed graph, 303, 314
 - application in 2-satisfiability, 314
 - over countably infinite sets, 112–113
 - over finite sets, 111–112
 - of partial orders, 111
- Total correctness, 145
- Total function, 19, 110
- Total order, 8, 12, 32, 395, 429, 434
 - least-significant digit sort, 33–34
 - lexicographic order, 33
 - sequence comprehension using, 33
- Towers of Hanoi, 422–424, 453–456
- Transitive closure, 31–32, 76, 116, 291, 317–325
 - all-pair shortest path algorithm, 321
 - compiling a metro connection directory, 321–324
 - of graph, 317–318
 - and matrix multiplication, 318–319
 - over semiring, 324–325
 - Warshall’s Algorithm, 319–321
- Transitive operators, omitting parentheses with, 42
- Transitive relation, 28–29
- Transitivity, 16, 29, 41, 56
- Transposition, permutation using, 101–102
- Tree, 9–10, 274–275
 - breadth-first traversal of, 295
 - depth-first traversal of, 296–297
 - free, 275
 - rooted, 274
- Tree isomorphism, 428–429
 - isomorphism of expressions, 430–431
 - isomorphism of expressions under idempotence, 431–432
 - simple isomorphism, 429–430
- Tree structures, 424
 - binary tree, 424–425
 - bst, 425
 - full binary tree, 425–427
 - general tree, 427–428
- Trimino tiling, 90–91
- Truth table, 38–40
- Truth-value assignment in 2-satisfiability, 316–317
- Tukey, J. W., 475
- Tuple, 7–8, 27, 33–34, 120, 382, 427
- Turing machine, 178, 377
- Unary minus in Haskell, 380, 384–385
- Uncomputable functions, 19
- Undirected dynamic graph, 305–311

- union-find algorithm
 - amortized cost analysis, [310–311](#)
 - description of the algorithm, [306–307](#)
 - optimization heuristics, [307–309](#)
 - properties of ranks, [309–310](#)
- Undirected graph, [266–269](#), [271](#), [283](#)
 - breadth-first traversal of, [295](#)
 - connected components of, [304](#)
 - depth-first traversal of, [303](#)
- Union of sets, [12–13](#), [15–16](#), [26](#), [34](#), [66](#), [75](#), [82](#)
- Union-find algorithm, [30](#), [177](#), [305–311](#), [346](#)
- Unipolar literal, [50](#)
- Unique prime factorization theorem, [82](#), [104–106](#)
- Unique representative of equivalence class, [30–31](#), [306](#), [429](#)
- Unit propagation, DPLL algorithm, [50](#)
- Universally quantified proof, [56](#)
- Unsorted lists, searching, [183–184](#)
- Vacuous relation, [25–26](#)
- Valiant, G., [250](#)
- Validity of logical formula, [44–50](#)
- van Heijenoort, J., [11](#), [63](#)
- van Leeuwen, J., [310](#)
- Variant function, [141–142](#), [144–145](#), [172](#)
- Venn diagram, [16–17](#)
- Verification, [139](#)
 - conditions, [164](#)
- Verified Software Initiative (VSI), [192–193](#)
- Vertex, [125](#), [266](#)
- VSI. *See* Verified Software Initiative (VSI)
- Vyssotsky, Vic, [370](#)
- Warshall's algorithm, [319–321](#)
 - all-pair shortest path algorithm, [321](#)
 - compiling a metro connection directory, [321–324](#)
 - for metro connection directory, [324](#)
 - over semiring, [324](#)
 - space requirement for, [320–321](#)
- Warshall, S., [319](#)
- Weak induction principle over natural numbers, [85–86](#)
- Weakening predicates, [42](#)
- Well-founded induction. *See* Noetherian induction
- Well-founded relation, [116](#)
 - equivalence of two notions of well-foundedness, [118](#)
 - examples, [116–117](#)
 - examples of application of well-founded induction, [119–121](#)
 - lexicographic order, [117–118](#)
 - well-founded induction principle, [118–119](#)
- where clause in Haskell, [387](#), [390](#)
- while command, [149](#)
- Wildcard, Haskell pattern matching, [398](#)
- Williams, J. W. J., [215](#)
- Winning strategy
 - in finite 2-player game, [113–115](#)
 - inductive proof of existence of, [115](#)
- Winograd, S., [497](#)
- Writing proofs, style of, [56–59](#)
- zChaff, [178](#)
- Zero-one principle, [480–481](#)
- zip constructor, [461–463](#), [493](#), [494](#)

Effective Theories in Programming Practice

Jayadev Misra

Set theory, logic, discrete mathematics, and fundamental algorithms (along with their correctness and complexity analysis) will always remain useful for computing professionals and need to be understood by students who want to succeed. This textbook explains a number of those fundamental algorithms to programming students in a concise, yet precise, manner. The book includes the background material needed to understand the explanations and to develop such explanations for other algorithms. The author demonstrates that clarity and simplicity are achieved not by avoiding formalism, but by using it properly.

The book is self-contained, assuming only a background in high school mathematics and elementary program writing skills. It does not assume familiarity with any specific programming language. Starting with basic concepts of sets, functions, relations, logic, and proof techniques including induction, the necessary mathematical framework for reasoning about the correctness, termination and efficiency of programs is introduced with examples at each stage. The book contains the systematic development, from appropriate theories, of a variety of fundamental algorithms related to search, sorting, matching, graph-related problems, recursive programming methodology and dynamic programming techniques, culminating in parallel recursive structures.

ABOUT ACM BOOKS



ACM Books is a series of high-quality books published by ACM for the computer science community. ACM Books publications are widely distributed in print and digital formats by major booksellers and are available to libraries and

library consortia. Individual ACM members may access ACM Books publications via separate annual subscription.

BOOKS.ACM.ORG · WWW.MORGANCLAYPOOLPUBLISHERS.COM

